Fachhochschule Münster

Fachbereich Elektrotechnik und Informatik

Diplomarbeit

Entwicklung eines Programms zur automatischen Analyse von C-Compilern hinsichtlich des C-Standards

von

Ralf Jonkmann

Referent:	Herr Prof. Dr. rer. nat. Nikolaus Wulff
Korreferent:	Herr Dipl.-Inf. Frank Böhland
Durchgeführt bei:	Robert Bosch GmbH, Leonberg
	Automotive Electronics - Body Engineering (AE-BE/ENG3)
	Postfach 1661, 71226 Leonberg

Kurzfassung

Durch die Verwendung verschiedener C-Compiler und Mikroprozessoren kann sich das Verhalten von Programmen ändern und damit Fehlfunktionen verursachen. Ursache ist eine unterschiedliche Implementierung bestimmter Eigenschaften der Programmiersprache C, welche in dem internationalen C-Standard ISO/IEC 9899 nicht oder nicht eindeutig definiert wurden. Ziel dieser Arbeit ist die Entwicklung eines Programms zur automatischen Analyse der nicht eindeutig definierten Eigenschaften im Hinblick auf eingebettete Systeme. Dieses Programm hilft Softwareentwicklern, Unterschiede zwischen Compilern besser zu erkennen, so dass im Vorfeld potenzielle Fehlerquellen erkannt werden und sich darauf basierende Fehler vermeiden lassen.

Danksagung

An dieser Stelle möchte ich allen danken, die mich während meines Studiums und dieser Diplomarbeit unterstützt haben.

Ich möchte der Robert Bosch GmbH und den Mitarbeitern der Abteilung AE-BE/ENG3[1] in Leonberg danken, die mir diese Diplomarbeit ermöglicht haben.

Ganz besonders bedanke ich mich bei Frank Böhland und Jost Brachert, die mich während meiner Diplomarbeit stets kompetent betreut und unterstützt haben. Sie gaben mehrfach Anregungen und Ratschläge, die zur erfolgreichen Durchführung der Diplomarbeit beigetragen haben.

Auch möchte ich mich bei Herrn Prof. Dr. Nikolaus Wulff für die Übernahme und dadurch die Ermöglichung der Diplomarbeit seitens der Fachhochschule Münster bedanken.

Meiner Freundin danke ich für das Korrekturlesen und die aufgebrachte Geduld.

Zu guter Letzt bedanke ich mich bei meinen Eltern, die mir meinen Ausbildungsweg ermöglicht haben. Sie standen mir jederzeit mit Rat und Tat zur Seite und gaben mir immer die nötige Motivation und moralische Unterstützung.

[1] Automotive Engineering - Body Electronics/Engineering 3

Inhaltsverzeichnis

Kurzfassung **I**

1 Einleitung **1**

1.1 Motivation 1

1.2 Aufgabenstellung 2

2 Hintergrund **4**

2.1 Eingebettete Systeme 4

2.2 Die Programmiersprache C 5

2.2.1 Ein kleiner Exkurs zum Build-Prozess 5

2.2.2 Der C-Standard 7

2.2.2.1 Implementation defined behavior 8

2.2.2.2 Unspecified behavior 8

2.2.2.3 Undefined behavior 8

2.3 Bekannte Ansätze 8

3 Zu analysierende Eigenschaften **10**

3.1 Analyse der implementationsabhängigen Eigenschaften 10

3.1.1 Translation 11

3.1.2 Environment 11

3.1.3 Identifiers 11

3.1.4 Characters 12

3.1.5 Integers 14

3.1.6 Floating point . . . 16
3.1.7 Arrays and pointers . . . 17
3.1.8 Registers . . . 17
3.1.9 Structures, unions, enumerations and bit-fields . . . 18
3.1.10 Qualifiers . . . 23
3.1.11 Declarators . . . 23
3.1.12 Statements . . . 23
3.1.13 Preprocessing directives . . . 24
3.1.14 Library functions . . . 24
3.2 Zusätzliche Eigenschaften . . . 25
3.2.1 Division und Modulo durch Null . . . 25
3.2.2 Common Extensions . . . 25
3.2.3 Makros . . . 25
3.2.4 Kommentare . . . 26

4 Programmsystem **27**
4.1 Build-Prozess . . . 27
4.2 Anforderungen an das Programmsystem *CEval* . . . 28
4.3 Datenmodell . . . 30
4.4 Lösungsansätze . . . 31
4.4.1 Flexibilität . . . 32
4.4.2 Compilerfehler . . . 33
4.4.3 Programmierung des Mikroprozessors . . . 33
4.4.4 Datenübertragung . . . 34
4.4.5 Kommunikation zwischen den Prozessen . . . 34
4.4.6 Dynamische Generierung von Quellcode . . . 35
4.4.7 Erweiterbarkeit . . . 35
4.4.8 Aufwand für den Anwender . . . 36

5 Implementierung **37**

5.1 Allgemeines . . . 37

5.1.1 Programmiersprache . . . 37

5.1.2 Entwicklungsumgebung . . . 38

5.2 Analysen . . . 38

5.2.1 Allgemein . . . 38

5.2.1.1 Namenskonvention . . . 38

5.2.1.2 Arten von Analysen . . . 39

5.2.1.3 Ausgabefunktionen . . . 40

5.2.1.4 Weitere Hilfsfunktionen . . . 41

5.2.1.5 Headerdateien . . . 41

5.2.2 Algorithmen zur Analyse der Eigenschaften . . . 42

5.2.2.1 Größe eines Byte . . . 42

5.2.2.2 Darstellung negativer Zahlen . . . 43

5.2.2.3 Datentypen . . . 43

5.2.2.4 Enum . . . 45

5.2.2.5 Modulo und Division von Integerzahlen . . . 46

5.2.2.6 Bitverschiebung . . . 47

5.2.2.7 Datentypkonvertierung . . . 48

5.2.2.8 Strukturen . . . 49

5.2.2.9 Endianess . . . 50

5.2.2.10 Bitfelder . . . 51

5.2.2.11 Gleitkommazahlen . . . 53

5.2.2.12 Zeichensatz . . . 56

5.2.2.13 Bezeichner . . . 58

5.2.2.14 Inhalt von Makros . . . 59

5.3 Automatisierung . . . 61

5.3.1 Anpassungen am Build-Prozess . . . 63

5.3.2 Der Programmlauf . . . 64

5.3.3 Das Programmsystem 65
5.3.3.1 Kommunikation 66
5.3.3.2 Konfiguration 67
5.3.3.3 Initialisierung 67
5.3.3.4 Generierung von Quellcode und Compilerfehleranalyse . . . 68
5.3.3.5 Die Auswertung 72
5.4 Review 73
5.5 Dokumentation 73

6 Anwendung und Ergebnisse **74**
6.1 Gleicher Prozessor, verschiedene Compiler 75
6.1.1 HC12 75
6.1.2 PowerPC 77
6.2 Unterschiedliche Prozessoren, verschiedene Compiler 77
6.3 Unterschiedliche Prozessoren, gleicher Compiler 78

Zusammenfassung **80**

Ausblick **82**

Eigenständigkeitserklärung **83**

Glossar **84**

Literaturverzeichnis **87**

Abbildungsverzeichnis **87**

A Portability Issues **88**

B Benutzerhandbuch **100**

C Ergebnisse **108**

Kapitel 1

Einleitung

Das „American National Standards Institute“ (ANSI) hat 1989 auf Grund der schon damals weiten Verbreitung der Programmiersprache C eine Standardisierung vorgenonmmen. Dieser als ANSI-C bekannte Standard wurde 1990 mit kleineren Änderungen als internationaler Standard ISO[1]/IEC[2] 9899[ISO90] festgelegt. Der Inhalt der beiden Standards ist identisch, nur der Aufbau unterscheidet sich [Wik07b][MIS04]. Er wurde 1993 originalgetreu in die europäische Norm EN 29899 übernommen. Im weiteren Verlauf wird der Einfachheit halber die allgemein bekannte Abkürzung C90 für diesen Standard verwendet. Zu dieser ersten Version gab es 1993 einen Zusatz. 1999 wurde der Standard komplett überarbeitet und in der neuen Version ISO/IEC 9899:1999[ISO99] herausgebracht, folgend als C99 bezeichnet. Auch wenn der C90 Standard schon mehr als 15 Jahre alt ist, wird er im Bereich von eingebetteten Systemen immer noch anstatt des neueren C99 angewandt.

1.1 Motivation

Als der größte Automobilzulieferer der Welt setzt die Firma Bosch in vielen Systemen elektronische Steuergeräte ein, in die Mikroprozessoren eingebettet sind. Je nach Anwendung werden unterschiedliche Prozessoren eingesetzt. Obwohl die Steuergeräte in ihrer Funktionalität völlig unterschiedlich sein können, sind viele Grundfunktionen der verwendeten Prozessoren gleich oder ähnlich, sodass sich auch die Software für diese Funktionen ähnelt. Aus diesem Grund ist es sinnvoll, wiederverwendbare Software zu entwickeln. Die Software für Mikroprozessoren wird häufig in C geschrieben und baut deshalb auf dem C-Standard auf.

[1] International Organization for Standardization
[2] International Electrotechnical Commission

In diesem Standard sind jedoch einige Eigenschaften „implementation defined“ und einige „undefined“. Die verschiedenen C-Compiler realisieren diese Eigenschaften entsprechend unterschiedlich. Weiterhin lässt sich das Verhalten von Compilern über Optionen einstellen. Welches Verhalten dabei geändert, wird ist auf den ersten Blick nicht immer ersichtlich. Diese beiden Unterschiede in verschiedenen Bereichen Auswirkungen haben:

- Oft werden Softwarekomponenten wiederverwendet, weil sie sich in der Anwendung bewährt haben und schon ausgiebig getestet wurden. Diese Tests wurden unter bestimmten Bedingungen durchgeführt. Um sicher zu stellen, dass eine Komponente das gewünschte Verhalten aufweist, muss bestätigt werden, dass diese Bedingungen auch bei dem neuen System erfüllt sind.

- Manchmal werden während der Softwareentwicklung die Compileroptionen geändert. Dadurch kann sich das Verhalten des Programms ändern. Viele Compiler bieten mehrere Arten von Optimierungen an. Der Maschinencode lässt sich beispielsweise auf Größe optimieren. Dadurch wird weniger Speicher verwendet und es können Kosten gespart werden. Welche genauen Auswirkungen diese Optimierungen haben, ist je nach Compiler unterschiedlich und nicht immer ersichtlich.

- Bei Bosch wechseln Entwickler häufig zwischen Projekten. Oft werden in den Projekten verschiedene Compiler verwendet. Damit von vornherein Fehler durch falsche Annahmen vermieden werden können, deren Entdeckung zu einem späteren Zeitpunkt Aufwand und Kosten verursachen würde, ist es für den Entwickler wichtig, von Beginn an die Unterschiede zu vorher genutzten Compilern zu kennen.

- Bei Bosch wird zur Überprüfung des Quellcodes das Programm QA-C[3] eingesetzt. Dieses Programm untersucht den Code auf kritische Programmkonstrukte und gibt entsprechende Warnungen aus. Um QA-C effektiv nutzen zu können, sind gewisse Grundeinstellungen bzgl. des jeweiligen Compilers / Prozessors notwendig. Sind diese Einstellungen nicht korrekt, werden falsche Warnungen ausgegeben und Fehler werden übersehen oder korrekter Code wird auf Grund von falschen Warnungen geändert.

1.2 Aufgabenstellung

Es ist ein Programm zu entwickeln, welches dem Softwareentwickler die Möglichkeit gibt, schnell und größtenteils automatisch die für die Programmierung von Mikroprozessoren wichtigsten, implementationsabhängigen Eigenschaften des C-Standards zu analysieren.

[3] QA-Systems, `http://www.qasystems.de`

Dadurch soll der Entwickler eine Übersicht über mögliche Fehlerquellen erhalten und in der Anwendung von QA-C unterstützt werden. Ferner sollen so die Auswirkungen von Compileroptionen besser sichtbar gemacht werden. Beispielsweise ist die Größe des Datentyps „int“ und die maximale Länge von Bezeichnern nicht festgelegt.
Das Programm soll so aufgebaut werden, dass die vom Anwender durchzuführenden Einstellungen minimal sind und sich nahe an einem der üblichen Build-Prozesse bewegen (Batchfile, Make). Dadurch soll die Anwendung des Programmsystems so einfach wie möglich gehalten werden. Das Programm kann somit ohne große Anpassungen in schon bestehenden Projektumgebungen eingesetzt werden. Dies ist notwendig, damit das Programm unter den gleichen Voraussetzungen (z.B. Compileroptionen) läuft, unter denen auch die zu entwickelnde oder portierende Software laufen soll.

Kapitel 2

Hintergrund

Im folgenden Kapitel werden die verschiedenen Aspekte der Aufgabe betrachtet.

2.1 Eingebettete Systeme

Heutzutage wird in fast allen Bereichen des täglichen Lebens Elektronik eingesetzt. Sei es in Waschmaschinen, Küchengeräten, Unterhaltungselektronik, Kraftfahrzeugen oder anderen Maschinen. Dies geschieht meist unsichtbar für den Anwender. Man spricht in diesem Fall von „Eingebetteten Systemen“ (engl. embedded systems).
Mit diesem Begriff werden Rechnersysteme bezeichnet, die integraler Bestandteil eines größeren Systems sind und meist nur für eine bestimmte Aufgabe konzipiert sind. Im Gegensatz zum normalen PC werden eingebettete Systeme speziell für ihren Aufgabenbereich entwickelt und optimiert. Dies macht sie effizienter, zuverlässiger und kostengünstiger. Je nach Anforderung werden die verschiedenen Aufgaben in Hardware oder in Software implementiert. Die Software unterscheidet sich stark von normaler PC-Software. „Kleine“ Systeme haben z.B. kein Betriebssystem. Weiterhin können eingebettete Systeme, wenn sie fertig entwickelt und in das Produkt eingebaut sind, oft gar nicht oder nur noch mit großem Aufwand geändert werden. Anders als beim PC lässt sich also eine fehlerhafte Software nicht kostengünstig durch ein Update oder Patch korrigieren, zumal dann üblicherweise eine große Anzahl an Geräten betroffen ist.
Auch die Firma Bosch verwendet in vielen Produkten eingebettete Systeme. Gerade wenn in kritischen Systemen, wie z.B in einem Motorsteuergerät oder im Steuergerät für das ABS[1], ein Softwarefehler auftritt, kann dies zu Rückrufaktionen führen, die immense Kosten verursachen oder aber im schlimmsten Fall zu Unfällen führen. Deshalb werden die Steuergeräte und deren Software vor der Markteinführung ausgiebigen Tests unterzogen.

[1] Anti Blockier System

Dieses Programm soll dazu beitragen, Fehler auf Grund von falschen Annahmen schon bei der Entwicklung zu vermeiden.
Ein weiterer Unterschied zu PC-Software ist, dass die Entwicklung der Software nicht auf dem System stattfindet, auf dem sie später zum Einsatz kommt. Es kommen so genannte Crosscompiler zum Einsatz. Diese laufen z.B. auf einem PC und generieren den für den verwendeten Mikroprozessor passenden Maschinencode, der dann im Mikroprozessor gespeichert wird. Die Entwicklungsumgebung (source environment) unterscheidet sich also von der endgültigen Umgebung (execution environment). Oft kann die Software während der Entwicklungsphase erst sehr spät auf dem tatsächlichen System getestet werden. Frühere Tests werden häufig mit Simulationen oder speziellen Entwicklungsboards durchgeführt. Diese können aber meist nicht alle Aspekte simulieren, was eine weitere Fehlerquelle darstellt.

2.2 Die Programmiersprache C

2.2.1 Ein kleiner Exkurs zum Build-Prozess

Jemand, der die Programmiersprache C kennenlernt, fängt wie bei jeder Sprache klein an. Das heißt, er lernt die Syntax kennen, die Struktur und schreibt schnell das bekannte „Hello World!“-Programm. Dieses wird kompiliert und es entsteht dann ein ausführbares Programm, welches „Hello World!“ auf den Bildschirm schreibt. Solange man sich mit kleinen Programmen und Übungen beschäftigt, bleiben einem wichtige Schritte auf dem Weg vom Quellcode zum fertigen Programm verborgen. Der Standard unterteilt diesen Weg in acht Teile, den sogenannten „Translation Phases“. In diesem Zusammenhang reicht aber auch die Unterteilung in drei Phasen:

1. Präprozessor
   Der Präprozessor übersetzt den Quellcode in den Standardzeichensatz, ersetzt Makros und bindet Code aus den über `#include` angegebenen Dateien ein.

2. Kompilieren
   Der Compiler überprüft die Syntax und übersetzt den Quellcode in Maschinensprache.

3. Linken
   Der Linker löst externe Abhängigkeiten auf. Funktionen und Variablen, die im aktuellen Modul zwar benutzt, aber nicht definiert werden, müssen über andere Module oder Bibliotheken eingebunden werden. Das Ergebnis ist das fertige, ausführbare Programm.

Bei größeren Softwareprojekten wird der Quellcode auf viele Dateien aufgeteilt. Da der letzte Schritt, das Linken, normalerweise erst möglich ist, wenn schon große Teile der Software fertig sind, werden die einzelnen Dateien (Module) zuerst nur kompiliert. Das heißt, es werden Objektdateien generiert und eventuell zu Bibliotheken zusammengefasst. Auch wenn jedes Modul für sich kompiliert werden kann, kann es zum Schluss beim Linken zu Problemen kommen.

Beispiel 1 Ein einfacher Fall in diesem Zusammenhang ist die doppelte Definition einer Funktion. Zwei Module können demnach jeweils eine Funktion mit dem gleichen Namen definieren. Solange diese Module nur für sich kompiliert werden, fällt dieses nicht auf. Erst beim Linken kann es zu Problemen kommen. Entweder es kommt zu einem Fehler, weil eine Funktion doppelt definiert wurde oder aber das zweite Modul wird nicht gelinkt, weil die Abhängigkeit schon durch das erste erfüllt wurde. Was genau geschieht, wird nicht durch den Standard festgelegt, sondern vom jeweiligen Compiler-Hersteller entschieden.

In jeder dieser drei Phasen können sich verschiedene Compiler unterschiedlich verhalten. Dies ist darauf zurückzuführen, dass im Standard nicht alles definiert wird und viele Punkte mit Absicht offen gelassen wurden, siehe Abschnitt 2.2.2.

Beispiel 2 Ein etwas komplexeres Problem, mit dem auch erfahrene C-Programmierer nicht unbedingt rechnen, soll der folgende Programmausschnitt zeigen. Hierbei geht es um die implizite Konvertierung von Datentypen. Der Standard legt fest, dass bei jeder arithmetischen Operation beide Operanden den gleichen Datentyp haben müssen. Das heißt, bevor gerechnet wird, werden die Operanden intern auf den gleichen Typ konvertiert. Dieser Vorgang wird auch Balancing genannt.

```
1  ...
2  int y=15;
3  if( y - 0x8000  < 0)
4  {
5          /* weitere Anweisungen */
6  }
7  ...
```

Auf den ersten Blick denkt man, dass der Ausdruck in Zeile 3 auf jeden Fall wahr ist. Mathematisch ist das jedenfalls korrekt. Die binären Operationen „minus“ und „kleiner als“ sind auch im Standard C90 klar definiert. Das Problem hier liegt in der Zahl `0x8000`. Allgemein werden alle Zahlen, die in ein `signed int` passen, dahin umgewandelt (Integer

Promotion). Passt der Wert nicht in ein `signed int`, gibt es verschiedene Möglichkeiten. Bei Konstanten hängt der Datentyp nicht nur von der Größe, sondern auch von dem Zahlenformat ab. Hexadezimal- und Oktalzahlen werden anders als Dezimalzahlen eingestuft. Dezimalzahlen werden in diesem Fall direkt zu `signed long` umgewandelt. Bei den beiden anderen kommt zuerst `unsigned int` und dann `signed long`. Ist `int` 32 Bit groß, gibt es kein Problem, da `0x8000` mit 32 Bit auf jeden Fall darstellbar ist.
Ist `int` jedoch nur 16 Bit groß, ist der Wert der Wert zu groß.[2] Deswegen wird `0x8000` in `unsigned int` umgewandelt. Jetzt greift das Balancing und wandelt das `y` auch zu `unsigned int`. Das Ergebnis des Ausdrucks `y - 0x8000` ist deshalb ebenfalls `unsigned int`, also positiv. Somit ist der `if`-Ausdruck immer falsch. Es findet noch ein zweites Balancing für den „kleiner als“ Operator statt, das aber keine Auswirkung mehr auf das Ergebnis hat.

Diese beiden Beispiele sollen einen kurzen Einblick in die vielfältigen Bereiche geben, in denen nicht klar definiertes Verhalten eine Rolle spielt. Gerade das zweite Beispiel zeigt, dass bei Software, die auf verschiedenen Mikroprozessoren laufen und mit verschiedenen Compilern kompiliert werden soll, auf viele, zum Teil auch versteckte, Eigenheiten geachtet werden muss.

2.2.2 Der C-Standard

Wie in der Einleitung erwähnt, gibt es verschiedene Versionen des C-Standards. Diese Arbeit behandelt hauptsächlich die Version aus dem Jahr 1990, ISO/IEC 9899 (C90). Auch wenn es eine überarbeitete Version aus dem Jahr 1999 (C99) gibt, wird heute bei eingebetteten Systemen noch C90 vorausgesetzt.
Viele Punkte des neueren Standards C99 werden von den Compilern bislang nicht unterstützt. Selbst der aktuellste C-Compiler der GNU Compiler Collection (GCC, v4.2) [GCC07] unterstützt noch nicht alle Neuerungen des C99. Generell garantieren viele kommerzielle Compilerhersteller, wie zum Beispiel Cosmic und Greenhills, für eingebettete Systeme bis heute nur die Einhaltung des C90 Standards [Cos07] [Gre07].
Im Oktober 2004 brachte ein Verband der Automobilindustrie (MISRA[3]) eine Richtlinie zur C-Programmierung heraus. Diese erwähnt, dass es zum Erscheinungsdatum keinen kommerziellen C-Compiler für eingebettete Systeme gab, der C99 unterstützt.[MIS04]

Die Schwierigkeit bei der Wiederverwendung von Software liegt an den vielen unterschiedlichen Compilern, die es für die Programmiersprache C, gibt und der großen Anzahl an verschiedenen Mikroprozessoren. Der Standard fördert zwar die Portierbarkeit, kann sie

[2] Wertebereich von 16-Bit signed int (Zweierkomplement): [-32768, 32767], 0x8000 = 32768.
[3] The Motor Industry Software Reliability Association

aber nicht garantieren. Um die nötige Flexibilität zu behalten, gibt es drei Beschreibungen für Eigenschaften, die nicht klar definiert werden können oder generell offen bleiben sollen.

2.2.2.1 Implementation defined behavior

Mit „implementation defined behavior“ wird ein korrektes Verhalten beschrieben, das aber von den Eigenschaften der jeweiligen Implementierung abhängt. Wie sich das Programm genau verhält, ist vom Hersteller des Compilers zu dokumentieren [ISO90]:3.10. Ein Beispiel für implementation defined behavior ist, wie im Beispiel 2 in 2.2.1 erwähnt, die Größe der einzelnen Datentypen.

2.2.2.2 Unspecified behavior

Hiermit wird Verhalten beschrieben, das zwar korrekt ist, auf welches der Standard aber nicht näher eingeht [ISO90]:3.17, wie zum Beispiel die interne Darstellung von Gleitkommazahlen [ISO90]:6.1.2.5.

2.2.2.3 Undefined behavior

Undefined behavior beschreibt nicht portable oder fehlerhafte Konstrukte, für die es im Standard keine Regeln gibt. Auch das Verhalten, wenn bestimmte, im Standard definierte Bedingungen nicht erfüllt sind, ist im Allgemeinen nicht definiert. Das heißt, der Standard verlangt nicht, dass es eine Dokumentation zu der jeweiligen Eigenschaft gibt, die beschreibt, wie sich das Programm in einem solchen Fall verhält. Es liegt also im Ermessen des Compilerherstellers, ob und wie diese Eigenschaften behandelt werden [ISO90]:3.16.

2.3 Bekannte Ansätze

Die Programmiersprache C ist gerade im Bereich von eingebetteten Systemen sehr verbreitet, da mit ihr hardwarenahe, schnelle und vor allem kleine Software erstellt werden kann. Jedoch zeichnet sich C nicht gerade durch Striktheit aus. Viele Konstrukte werden zugelassen, die bei anderen Programmiersprachen zu Fehlern führen würden. Typen werden zum Beispiel implizit umgewandelt, die Größe von Arrays nicht überprüft und vieles mehr. Deshalb ist gerade bei kritischer Software besondere Vorsicht geboten. Es gibt dementsprechend in der Atomindustrie, der Luftfahrt und anderen Industriezweigen zusätzliche Regeln für die Erstellung von Software mit C.
Auch die Automobilindustrie hat sich zu diesem Zweck zusammengeschlossen und ein Regelwerk entworfen. Diese Richtlinie (MISRA-C:2004) enthält 121 notwendige Regeln und

20 Hinweise im Umgang mit C. In der Richtlinie steht jedoch, dass es nicht praktikabel ist, alle Regeln ohne Ausnahme zu erfüllen. Notwendig heißt deshalb in diesem Zusammenhang, dass alle Abweichungen von diesen Regeln dokumentiert werden müssen, um MISRA konformen Code zu erstellen. [MIS04]
Zu diesen Regeln, sei es MISRA oder ein anderes Regelwerk, gibt es auch entsprechende Software, die den entwickelten Quellcode auf die Einhaltung dieser Regeln hin überprüft, wie z.B. QA-C[4] und PC-lint[5]. Diese Software beinhaltet meist noch weitere mehr Regeln, die auf allgemeinen Erfahrungen basieren. Das bei Bosch eingesetzte Programm QA-C benötigt für seine Tests bestimmte Informationen über den verwendeten Compiler und den Prozessor. Dabei handelt es sich z.T. um die Angaben, die auch im Standard nicht genau definiert sind. An dieser Stelle soll das von mir entwickelte Programm ansetzen und dem Benutzer in kurzer Zeit diese Informationen zur Verfügung stellen.

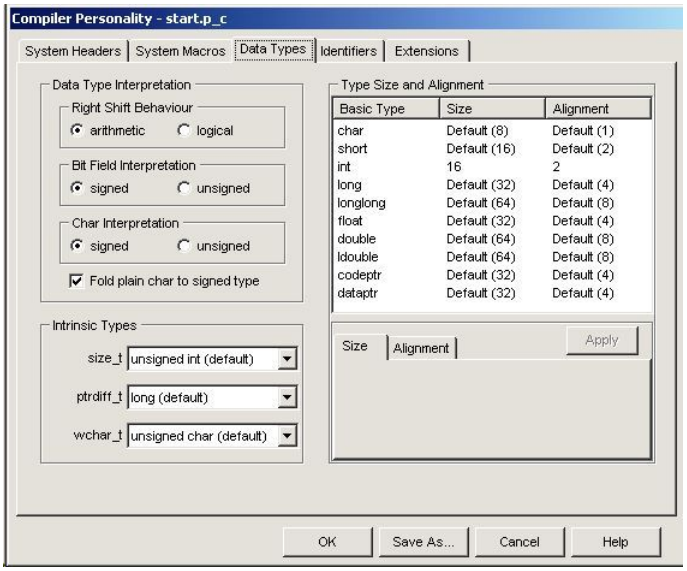


Abbildung 2.1: Konfigurationsmenü von QA-C

Abbildung 2.1 zeigt einen Ausschnitt aus dem Konfigurationsmenu von QA-C. Hier muss unter anderem die Größe der verschiedenen Datentypen angegeben werden.

[4]QA-Systems, `http://www.qasystems.de`
[5]Gimpel Software, `http://www.gimpel.com`

Kapitel 3

Zu analysierende Eigenschaften

Im folgenden Kapitel wird auf die Eigenschaften der Programmiersprache C eingegangen, die von dem Programm geprüft werden sollen. Diese basieren hauptsächlich auf den „implementation defined features“ des Standards C90. Hinzu kommen zusätzliche Konfigurationseinstellungen des Programms QA-C, Anregungen der Entwickler bei Bosch und einige Punkte der „Common Extensions“ des Standards C90.
Neben den Auswirkungen, die eine bestimmte Eigenschaft auf ein Programm haben kann, wird auch erklärt, warum einige Eigenschaften nicht geprüft werden. Auf die Tests und Algorithmen, mit denen das genaue Verhalten festgestellt werden kann, wird in Kapitel 5 näher eingegangen.
Vorab kann gesagt werden, dass nur Eigenschaften überprüft und analysiert werden, die für eingebettete Systeme relevant sind. So werden zum Beispiel Funktionen zur Bearbeitung von Dateien und zum Zugriff auf Dateisysteme vollständig ausgelassen.

3.1 Analyse der implementationsabhängigen Eigenschaften

Der Anhang G des Standards C90 befasst sich mit den „Portability Issues“ der Programmiersprache C. Es ist eine elfseitige, stichpunktartige Zusammenfassung der offenen Punkte, unterteilt in eine Seite „unspecified behavior“, viereinhalb Seiten „undefiened behavior“ und drei Seiten „implementation-defined behavior“. Die restlichen Seiten befassen sich mit „Common Extensions“, also Erweiterungen, die nicht durch den Standard abgedeckt sind, aber allgemein üblich sind. Die komplette Liste der „Portability Issues“ befindet sich im Anhang A. Die folgenden Abschnitte behandeln die implementationsabhängigen Eigenschaften, die im „Annex G.3“[ISO90] zusammengefasst sind. Die Überschriften sind dem „Annex G.3“ entnommen.

3.1.1 Translation

Mit Translation sind die acht Phasen der Übersetzung eines Programms gemeint. Diese können vom Compilerhersteller zusammengefasst werden. In diesem Zusammenhang kann ein Hersteller Diagnosemeldungen selbst definieren. Der Standard gibt nur vor, dass bei einem Fehler eine Meldung erzeugt werden muss. Wie diese auszusehen hat, ist nicht festgelegt ([ISO90]:5.1.1.3).
Da der Fantasie der Compilerentwickler in diesem Punkt keine Grenzen gesetzt sind, wird auf diesen Punkt nicht näher eingegangen. Wie im Programmablauf trotzdem Compilerfehler erkannt werden, wird in 4.4.2 erklärt.

3.1.2 Environment

Dieser Punkt geht auf die Umgebung einer Anwendung ein, zum einen wie Argumente, an die Funktion `main` übergeben werden ([ISO90]:5.1.2.2.1), und zum anderen, was ein interaktives Device ausmacht. Der erste Punkt ist für eingebettete Systeme nicht interessant, da ein Programm normalerweise nicht interaktiv aufgerufen wird.
Weiterhin ist es generell nicht möglich zu wissen, welche internen oder externen Komponenten ein beliebiger Mikroprozessor ansteuern kann und wie dieses geschieht. Das heißt, ob und wie der Prozessor mit seiner Außenwelt kommunizieren kann, hängt nicht mit dem Standard zusammen und wird deshalb auch nicht näher betrachtet.

3.1.3 Identifiers

Identifier oder Bezeichner müssen vom Präprozessor, Compiler und Linker nur bis zu einer bestimmten Länge unterschieden werden. Die minimal unterscheidbare Länge von Bezeichnern legt der Standard auf 31 Zeichen für interne (lokale) und 6 Zeichen für externe (globale) fest ([ISO90]:6.1.2). Mit Bezeichnern sind alle Namen gemeint, die vom Programmierer frei gewählt werden können. Das geht von Variablen-, Funktions-, Label- und Makronamen bis zu eigenen Typennamen und den einzelnen Elementen von Strukturen, Unions und Enumerations. Dass für externe Bezeichner nur sechs signifikante Stellen vorausgesetzt werden, liegt daran, dass es zur Zeit der Festlegung des Standards Implementierungen mit diesem Limit gab. Deshalb wird in dem Unterpunkt „Future language directions“ des Standards dieses Limit auch auf 31 Zeichen erhöht ([ISO90]:6.9.1).
In vielen Softwareprojekten gibt es Namenskonventionen, um sprechende und eindeutige Bezeichner zu erzeugen. Es werden auch immer mehr Programme eingesetzt, die automatisch Code generieren. Beides führt dazu, dass die garantierten 31 Zeichen schnell überschritten sind. Eine Namenskonvention könnte zum Beispiel so aussehen:

```
Projektname_Modulname_sprechenderFunktionsname()
```

Dieser Bezeichner hat schon 46 Zeichen. Die meisten Compiler unterscheiden deshalb längere Bezeichner.
Bei der Überprüfung der tatsächlich akzeptierten Länge gibt es verschiedene Szenarien zu beachten. Kommt es bei internen Variablen zu Überschneidungen, weil sie zu lang sind und sich nur in den signifikanten Stellen nicht unterscheiden, wird der Compiler dieses direkt erkennen und einen Fehler ausgeben.
Bei externen Variablen kann es aber sein, dass der Compiler einfach die letzten Stellen ignoriert und es erst beim Linker zu Problemen führt. So kann es sein, dass in zwei Modulen zwei Variablen mit fast gleichem Namen deklariert sind, die sich in den signifikanten Stellen nicht unterscheiden. Beim Linken kann es nun passieren, dass der Linker dieses merkt und eine Fehlermeldung ausgibt. Andererseits kann es aber auch sein, dass das andere Modul gar nicht eingebunden wird, da die Abhängigkeiten schon durch das erste Modul erfüllt sind und somit der Fehler nicht bemerkt wird.
Neben der signifikanten Länge der Bezeichner wird bei externen Bezeichnern ebenfalls nicht festgelegt, ob zwischen Groß- und Kleinschreibung unterschieden wird.

3.1.4 Characters

Zeichensatz

Generell unterscheidet der Standard zwischen dem Entwicklungssystem und dem System, auf dem das Programm letztendlich ausgeführt. Dementsprechend werden auch die geltenden Zeichensätze unterschieden. Diese werden auch als Quell- und Ausführungszeichensatz bezeichnet (siehe 2.1) und müssen nicht identisch sein ([ISO90]:6.1.3.4).
Der Standard legt fest, dass jedes C-System über die folgenden Zeichen verfügen muss. Die interne Darstellung wird jedoch nicht festgelegt.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1 2 3 4 5 6 7 8 9 0
! " % & / ( ) = ? { } [ ] \ * + ~ ^ ' # - _ : . ; , < > |
```

space, new line, horizontal tab

Für die folgenden Steuerzeichen muss der Quellzeichensatz Symbole zur Verfügung stellen, die dann während der Translation (siehe Abschnitt 3.1.1) in den entsprechenden Zeichencode für den Mikroprozessor umgewandelt werden, wie beispielsweise `\n` für *new line*.

`\0` *null*	`\t` *horizontal tab*	`\v` *vertical tab*	`\f` *form feed*
`\a` *alert*	`\n` *new line*	`\r` *carriage return*	`\b` *backspace*

Welche zusätzlichen Zeichen erlaubt sind, ist implementationsabhängig ([ISO90]:5.2.1). Mittlerweile wird fast nur noch der ASCII[1]-Zeichensatz eingesetzt. Neben diesem gibt es noch vereinzelt den EBCDIC[2] Zeichensatz. Im Zuge der Analyse reicht es aber, den Anwender darauf aufmerksam zu machen, wenn es sich bei dem verwendeten Zeichensatz nicht um ASCII handelt.
Weiterhin soll auch überprüft werden, ob der erweiterte ASCII-Zeichensatz akzeptiert wird.
Der Standard definiert für einige Zeichen Trigraphen (`??=` für `#`, `??(` für `[`, ...). Diese sind ein Relikt aus der Zeit, als Tastaturen noch nicht alle Zeichen unterstützten und werden nicht weiter überprüft.

Byte

Im Zusammenhang mit dem Zeichensatz steht auch die Größe des Datentyps `char`. Ein `char` repräsentiert in C ein Byte. Abweichend zur allgemeinen Verwendung des Begriffs Byte für eine Menge von 8 Bit, definiert der Standard C90 ein Byte als eine Gruppe von aufeinander folgenden Bits, mit der sich alle Zeichen des Basiszeichensatzes darstellen lassen. Weiterhin soll jedes Byte einzeln adressierbar sein. Es wird also nicht festgelegt, ob ein Byte 8, 9 oder mehr Bit hat. Die Mindestgröße liegt jedoch bei 8 Bit ([ISO90]: 5.2.4.2.1). Einige der ersten Mainframesysteme waren z.B. 36-Bit Maschinen mit einer Bytegröße von 9 Bit.[3] Heutzutage stellt ein Byte normalerweise 8 Bit dar. Es gibt aber auch Anwendungsbereiche, wie die digitale Signalverarbeitung, in denen es auf sehr hohen Datendurchsatz ankommt. Ein Beispiel für einen Compiler, dessen Datentyp `char` eine Größe von 16 Bit hat, ist der Compiler für die TMS320S55x Serie der Firma Texas Instruments [Tex03]. Dieses Merkmal ist jedoch so herausragend, dass der Mikroprozessor wahrscheinlich genau aus diesem Grund genutzt wird. Deshalb wird die Größe eines Bytes zwar überprüft, aber für alle anderen Tests mit 8 Bit vorgegeben. Der Anwender erhält eine Warnung, falls das Ergebnis nicht den erwarteten 8 Bit entspricht. In diesem Fall sind alle anderen Aussagen ungültig. Nach rücksprache wurde vereinbart, das Analysen für den Fall $Byte \neq 8\ Bit$ nicht relevant sind.

[1] American Standard Code for Information Interchange
[2] Extended Binary Code Interchange Format
[3] zum Beispiel die UNIVAC 1100 Serie

Char

Der einfache Datentyp `char` kann entweder als `signed char` oder als `unsigned char` interpretiert werden. Wird zum Beispiel mit Überläufen gerechnet, ist es wichtig, ob dieser schon bei 128 oder erst bei 256 auftritt oder ob negative Zahlen darstellbar sind.

Multibyte Characters

Neben dem Standardzeichensatz bietet C auch die Möglichkeit, andere Zeichen zu verwenden. Dazu sind sogenannte „multibyte charaters“ definiert. Der Datentyp für diese Zeichen ist `wchar` (wide character). Da deren Verwendung bei eingebetteten Systemen jedoch nicht praktikabel ist, wird im Rahmen dieser Analyse nicht weiter darauf eingegangen. Genauer wird dieser Datentyp auch erst in einer Änderung des Standards (ISO/IEC 9899/Amd.1) aus dem Jahre 1993 definiert.

3.1.5 Integers

Im Bereich der Integertypen gibt es ebenfalls eine Reihe von compilerabhängigen Eigenschaften. Dabei handelt es sich neben der Größe der einzelnen Datentypen hauptsächlich um den Umgang mit negativen Zahlen.

Größe und interne Darstellung der verschiedenen Datentypen

Der Standard legt für die verschiedenen Datentypen `short`, `int` und `long` nur Mindestgrößen fest. Für `unsigned int` liegen diese bei [0, 65535] und für `signed int` bei [-32767, 32767], also jeweils 16 Bit. Je nach Compiler kann ein `int` auch mit mehr Bits dargestellt werden, üblicherweise dann mit 32 Bit. Eine Auswirkung der unterschiedlichen Größe wurde in Abschnitt 2.2.1, Beispiel 2 erklärt. Wie groß die einzelnen Datentypen genau sind, soll durch die Analyse festgestellt werden.

Weiterhin wird auch die interne Darstellung von `signed` und `unsigned` Datentypen nicht genau festgelegt. Es wird nur gesagt, dass sich die Anzahl und Ausrichtung der Bits nicht unterscheiden darf ([ISO90]:6.1.2.5). Ob negative Zahlen im Zweierkomplement, Einerkomplement oder mit Vorzeichenbit gespeichert werden, wird nicht festgelegt. Da das Zweierkomplement die übliche Darstellung ist, wird diese für die weiteren Analysen vorausgesetzt. Es wird nur überprüft, ob dies auch der Fall ist. Sollte das Zweierkomplement nicht verwendet werden, wird eine Warnung an den Anwender ausgegeben, da andere Aussagen dann nur bedingt korrekt sind.

Die Auslegung der anderen Analysen auf die verschiedenen Darstellungen daher nicht notwendig. Da keine Mikroprozessoren mit einer anderen Darstellung zur Verfügung stehen oder überhaupt bekannt sind, könnten darauf basierende Analysen nicht getestet werden.

Datentypkonvertierung

In der Programmierung werden Werte oft von einem Datentyp in einen anderen umgewandelt (casts). Dies geschieht oft implizit, wie bei der Integer Promotion und beim Balancing (siehe Abschnitt 2.2.1), oder explizit durch den Softwareentwickler. Ob und wie sich der Wert verändert, hängt dabei davon ab, ob sich der Wert durch den neuen Datentypen darstellen lässt. Für einen Wert, der zu groß für den neuen Typen ist, wird das Verhalten nur festgelegt, wenn der Zieldatentyp `unsigned` ist. Der neue Wert ist der Rest der Division des ursprünglichen Wertes mit dem maximalen Wert des Zieldatentyps plus eins. Bei der internen Darstellung im Zweierkomplement verändert sich dabei kein Bit. Ist der Zieldatentyp jedoch `signed`, bestimmt die Implementierung, wie ein zu großer Wert konvertiert wird ([ISO90]:6.2.1.2).
Wichtig für den Softwareentwickler ist dabei, ob sich das Bitmuster der internen Darstellung ändert oder das Muster beibehalten wird.

Division und Modulo

Die Integer Division hat immer ein ganzzahliges Ergebnis. Modulo ergibt den Rest der Integer Division. Die folgende Formel stellt den Zusammenhang der beiden Operationen dar:

$$(a/b) * b + a\%b = a$$

Für positive Zahlen wird festgelegt, dass das Ergebnis der Division immer zur nächstkleineren Zahl gerundet wird. Dementsprechend ist auch das Ergebnis von Modulo festgelegt. Ist eine der beiden Zahlen negativ, ist es implementationsabhängig, ob das Ergebnis einer Division die nächste ganze Zahl kleiner gleich oder größer gleich dem Quotienten ist. Die folgenden Kombinationen erfüllen die Formel:

```
 9/ 5 =  1            =>     9% 5 = 4
 9/-5 = -1 oder -2    =>     9%-5 = 4 oder -1
-9/ 5 = -1 oder -2    =>    -9% 5 =-4 oder  1
-9/-5 =  1 oder  2    =>    -9%-5 =-4 oder  1
```

Bei der Division wird normalerweise das erste Ergebnis erwartet (± 1), sowie bei Modulo wenigstens der Betrag des ersten Ergebnisses (4), weshalb eine Überprüfung des implementierten Verhaltens wichtig ist .

Bitverschiebung

Im letzten Punkt des Integerabschnitts geht es um die Bitverschiebung. Auch hier handelt es sich nur um das Verhalten bei `signed` Werten. Die Frage ist, ob bei einer Rechtsverschiebung die Bits immer mit 0 oder mit dem Inhalt des höchstwertigen Bits aufgefüllt werden. In diesem Zusammenhang wird auch von einer logischen Verschiebung bzw. einer arithmetischen Verschiebung gesprochen.

```
1 signed int a = -1;
2 a = a >> 1;
```

Ob `a` nun immer noch -1 ist oder 32767, liegt am Compiler.[4] Wie schon erwähnt, wird das allgemein übliche Zweierkomplement vorausgesetzt und `int` ist in diesem Fall 16 Bit groß.

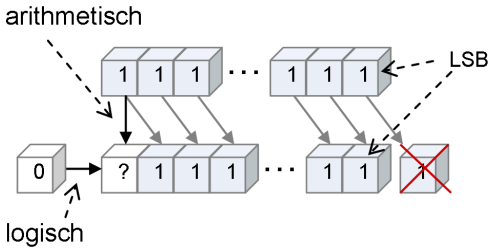


Abbildung 3.1: Rechtsverschiebung bei signed Werten

3.1.6 Floating point

Auch bei den drei Gleitkommatypen `float`, `double` und `long double` ist nur eine Mindestgröße angegeben ([ISO90]:6.1.2.5). Hier ist neben der Größe aber auch die maximale Genauigkeit interessant. Zurzeit werden Gleitkommazahlen eher selten in Steuergeräten eingesetzt, da für deren Berechnungen zu viel Zeit benötigt wird und deren Verwendung das Einbinden größerer Bibliotheks-Funktionen beinhaltet. Dies wird sich aber mit neuen, leistungsfähigeren Mikroprozessoren und neuen Anforderungen ändern. Deshalb ist es interessant, die Genauigkeit und Größe der einzelnen Typen zu kennen.

Im Gegensatz zur internen Darstellung von Integer-Typen ist die Darstellung der drei Gleitkommatypen `float`, `double` und `long double` nicht genau definiert und wird als „unspecified“ deklariert. Es wird nur die folgende mathematische Bedingung gegeben:

$$x = s * b^e * \sum_{k=1}^{p} f_k * b^{-k}$$

[4] $-1_{10} = 111...111_2$ (Zweierkomplement) und $32767_{10} = 011...111_2$

mit:

x = normalisierte Gleitkommazahl ($f_1 > 0, x \neq 0$)
s = sign (± 1)
b = Basis
e = Exponent (Integer zwischen $e_{min} \leq e \leq e_{max}$)
p = Präzision, Anzahl der Basis b Stellen im signifikanten Teil
f_k = nicht negatives Integer kleiner als b

Neben der Genauigkeit ist auch das Rundungsverhalten implementationsabhängig. Dabei werden zwei Fälle unterschieden. Einerseits, wenn zum Beispiel ein `long` zu einem `float` konvertiert wird und sich dessen Wert nicht genau darstellen lässt ([ISO90]:6.2.1.3), andererseits, wenn ein Gleitkommawert nicht genau dargestellt werden kann ([ISO90]:6.2.1.4). Der Standard gibt vier Rundungsverhalten (gegen Null, zum nächsten Nachkommastelle, gegen plus Unendlich und gegen minus Unendlich) vor, es steht dem Compilerhersteller aber offen, andere zu definieren.

3.1.7 Arrays and pointers

Die Größe eines Arrays wird mit Hilfe des Operators `sizeof` bestimmt. Dessen Rückgabewert ist eine Integerkonstante, deren Datentyp mit `size_t` bezeichnet wird. `size_t` ist kein neuer Datentyp, sondern ein anderer Name für einen der Standarddatentypen, und wird in einem der Standardheader definiert und dadurch implementationsabhängig. Laut Standard ist die maximale Größe eines Arrays begrenzt durch die Größe von `size_t` und deshalb ebenfalls implementationsabhängig ([ISO90]:6.3.3.4).
Wie groß ein Pointer ist, hängt zum Teil auch mit der Größe des verwendeten Speichers ab, also wie viel Speicher adressiert werden kann/muss. Dabei wird außerdem zwischen Code- und Datenpointern unterscheiden. Code wird meistens im ROM des Mikroprozessors gespeichert, während Daten zur Laufzeit im RAM abgelegt werden. Hier ist es interessant zu wissen, ob es unterschiedliche Pointertypen gibt. Dies gehört zwar direkt nicht zum Standard C90, kommt aber in der Praxis öfters vor und ist eine Einstellung von QA-C.
Auch der Datentyp der Differenz von zwei Pointern zu Elementen eines Arrays wird von dem Compilerhersteller bestimmt. Dieser wird `ptrdiff_t` genannt und wird auch in einem Standardheader definiert.

3.1.8 Registers

Bei `register` handelt es sich um einen „storage-class specifier“, der dem Compiler sagt, dass der Zugriff auf dieses Objekt so schnell wie möglich erfolgen soll. Inwieweit der Compilerhersteller dieses aber umsetzen muss, lässt der Standard völlig offen. Es war nicht

möglich, mit einfachen, automatisierbaren Methoden herauszufinden, ob und wie ein Compiler diese Eigenschaft implementiert. Dazu müsste zum Beispiel der Assemblercode analysiert werden, um zu sehen, wo im Speicher die mit `register` klassifizierten Variablen abgelegt werden. Eine Analyse des Assemblercodes ist aber nicht vorgesehen, da sich dieser auch je nach Compiler und Mikroprozessor unterscheidet. Deshalb wird dieser Punkt in der Analyse außer Acht gelassen.

3.1.9 Structures, unions, enumerations and bit-fields

Bei Strukturen und Unions handelt es sich um zusammengesetzte Datentypen. Dabei spielt besonders die Ablage der Datentypen im Speicher eine Rolle.

Unions

Unions bieten die Möglichkeit, mehrere verschiedene Datentypen auf dem gleichen Speicherbereich abzulegen. Oft werden sie bei der Mikroprozessorprogrammierung in Verbindung mit Strukturen dazu verwendet, auf einzelne Bits eines Registers[5] zuzugreifen, oder um das ganze oder mehrere Register auf einmal zu bearbeiten oder auszulesen. Unions bieten auch eine gute Möglichkeit, Speicher zu sparen. Da jedoch die Anordnung der Bytes der verschiedenen Datentypen von Prozessor zu Prozessor unterschiedlich sein kann, ist das Auslesen von Daten, die über ein anderes Element als dem aktuellen in die Union geschrieben wurden, implementationsabhängig. In diesem Zusammenhang spielt die Anordnung der Bytes eine große Rolle (Endianess). Es gibt hauptsächlich Little-Endian (z.B. Intel Prozessoren) und Big-Endian (z.B. Motorola Prozessoren). Abbildung 3.2 zeigt graphisch die Anordnung der Bytes anhand der Hexadezimalzahl 0x04030201.

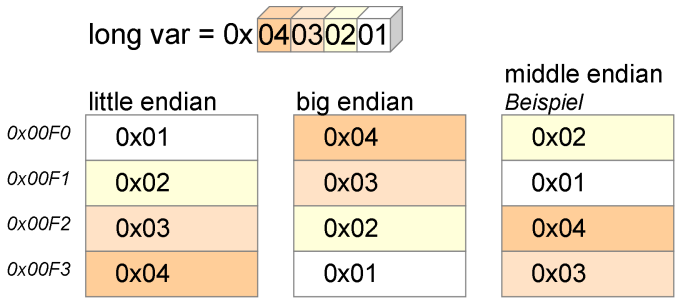


Abbildung 3.2: Anordnung von Bytes

Welche Auswirkungen die unterschiedliche Anordnung der Bytes auf Unions hat, zeigt das folgende Codebeispiel:

[5] Register bezeichnet hier Steuerregister oder I/O-Register, nicht die Registerbänke der CPU.

```
1 /* Voraussetzung: short ist 16 Bit groß */
2 union {
3   unsigned char var_uc;
4   unsigned short var_us;
5 } test_un;
6 test_un.var_us = 0x2010;
```

Je nach Endianess hat `test_un.var_uc` nun den Wert 0x20 bei Big-Endian oder 0x10 bei Little-Endian. Bei einigen Compilern lässt sich die Endianess umschalten (PowerPC, ARM, MIPS). Manche Prozessoren benutzen Big-Endian für die Anordnung der Bytes in einem Datenwort (16 Bit) und Little-Endian für die Anordnung der Datenworte (bei Datentypen mit mindestens 32 Bit). In diesem Fall wird von Middle-Endian oder Mixed-Endian gesprochen. [Wik07a]

Strukturen

In Strukturen werden Daten in Gruppen zusammengefasst. Dies bietet die Möglichkeit, zusammengehörige Daten als Gruppe zu übergeben. Solange dieses auf nur einem System stattfindet, stellt der Zugriff auf die Daten kein Problem dar. Kommunizieren aber unterschiedliche Systeme miteinander, spielt das Alignment[6], also die Anordnung der Speicheradressen, eine große Rolle. In vielen Systemen kann beispielsweise eine Variable vom Datentyp `int` nur an geraden Adressen abgelegt werden, wohingegen ein `char` an jeder beliebigen Adresse liegen kann. Werden diese Datentypen in einer Struktur kombiniert, kann dies dazu führen, dass Füllbytes genutzt werden, um die richtige Adressierung zu gewährleisten (Padding). Neben Zugriffsproblemen, wirkt sich das Verwenden von Füllbytes auch negativ auf den Speicherverbrauch aus.

```
1 struct{
2   char var_c;
3   int  var_i;
4 } test_str;
```

In diesem Beispiel belegt ein `int` 2 Byte. Links in der Abbildung 3.3 wird der interne Aufbau dargestellt, bei dem nur gerade Adressen für eine `int`-Variable erlaubt sind. Je nachdem, wie viele Bytes ein Datentyp belegt, kann es auch sein, dass die jeweilige Adresse durch 4 oder durch 8 teilbar sein muss. Dies kann dazu führen, dass Strukturen wesentlich mehr Speicherplatz benötigen als erwartet. Werden Strukturen nun von einem System abgelegt und von einem zweiten System mit anderem Alignment gelesen, kann dies zu

[6] Zahl, durch die die Speicheradresse eines Datentyps teilbar sein muss.

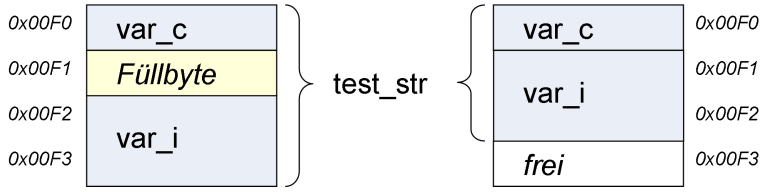


Abbildung 3.3: Alignment in Strukturen

Problemen führen. Es gibt auch Compiler, bei denen sich das Alignment über Optionen einstellen lässt.[7]

Bitfelder

Bei Bitfeldern handelt es sich um einen Spezialfall der Strukturen. Neben der Endianess, dem Alignment und Padding ist auch das Vorzeichen implementationsabhängig.
Bitfelder haben laut Standard den Datentyp `int`.

```
1 struct{
2    int bits1:6;
3    int bits2:16;
4    int bits3:10;
5 }bitFeld_str;
```

Allgemein wird das einfache `int` als `signed` interpretiert. Bei einem Bitfeld ist dies aber vom Compiler abhängig. `bitFeld_str.bits1` kann also je nach Compiler `signed` ([-32,31]) oder `unsigned` ([0,63]) sein.

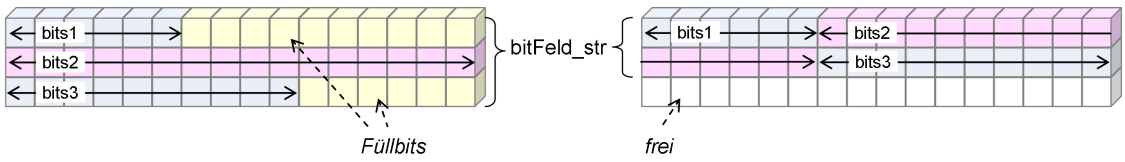


Abbildung 3.4: Padding in Bitfeldern

Weiterhin ist es implementationsabhängig, ob für die Struktur `bitFeld_str` zwei oder drei `int` belegt werden. Der Compiler kann ein Bitfeld über zwei Einheiten verteilen. In Abbildung 3.4 sind beide Möglichkeiten schematisch dargestellt. Im rechten Teil wird das Bitfeld `bits2` über zwei Einheiten verteilt, wohingegen im linken Teil Füllbits genutzt werden (Padding).

[7] z.B. beim Cosmic Compiler für den HC12 Mikroprozessor mit der Option +even. [Cos05]

Neben der Anordnung mehrerer Bitfelder ist auch die Reihenfolge der Bits nicht vorgegeben. Das heißt, es gibt auch auf Bitebene Endianess, die aber der Endianess auf Byte-Ebene nicht entsprechen muss. Je nach Ausrichtung liegt das erste Bit des Bitfeldes auf dem niederwertigstem Bit (LSB[8]) oder auf dem höchstwertigstem (MSB[9]) der Einheit.

```
1 struct {
2   int bits1:1;
3   int bits2:5;
4 }bitFeld_str;
```

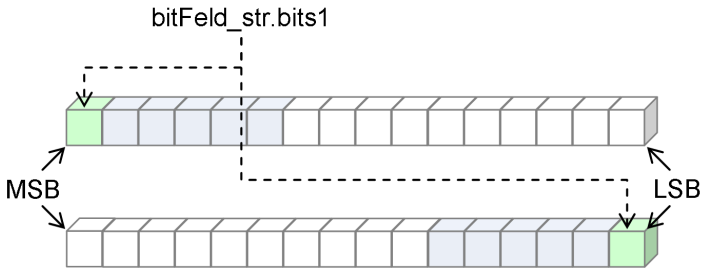


Abbildung 3.5: Anordnung der Bits

Viele Compiler bieten die Möglichkeit, Bitfelder zu optimieren, indem nur soviel Speicher reserviert wird, wie von den einzelnen Bits benötigt wird. Das heißt, obwohl ein Bitfeld als `int` definiert wird, wird es intern in einem `char` abgelegt. Wenn der Compiler den Datentyp ändert, weil nur wenige Bits verwendet werden, kann es sein, dass das erste Bit von zwei `int`-Bitfeldern an verschiedenen Positionen liegt, wie in der Abbildung 3.6 dargestellt.

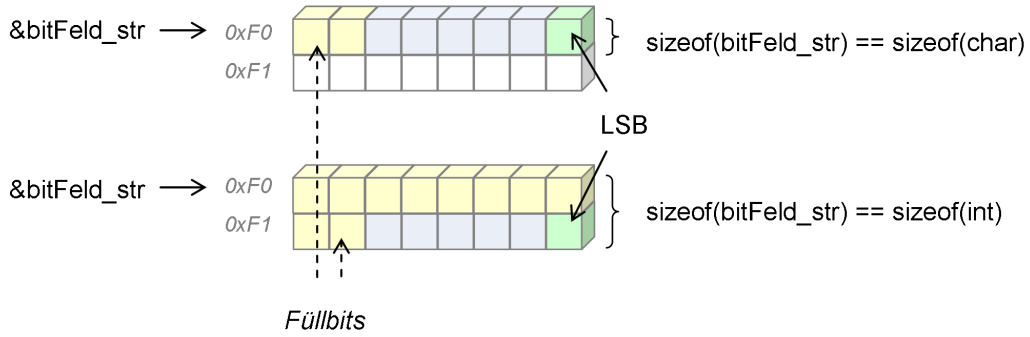


Abbildung 3.6: Verschiebung des LSB bei Big-Endian

[8]Least significant bit
[9]Most significant bit

Das Gleiche gilt auch für das MSB bei Little-Endian.

Enumeration

Mit dem Datentyp `enum` lassen sich in C Integer-Werte mit Namen belegen. Wenn Enumerations verwendet werden, wird intern dennoch mit Zahlen gearbeitet. Von welchem Integertyp die Enumerations sind, ist implementationsabhängig. Das heißt, der Compiler entscheidet, ob die Elemente eines Enums als `signed` oder `unsigned` Wert gespeichert werden und welche Größe der entsprechende Datentyp hat. Es gibt dabei unterschiedliches Verhalten, das anhand des folgenden Codebeispiels erklärt werden soll. Dabei wird angenommen, dass `int` 16 Bit groß ist.

```
1 enum en1 {a,      b,          c};     /* a=0,   b=1,       c=2       */
2 enum en2 {d,      e=400,      f};     /* d=0,   e=400,     f=401     */
3 enum en3 {g=-4,   h=3,        i};     /* g=-4,  h=3,       i=4       */
4 enum en4 {j=-2,   k=40000,    l};     /* j=-2,  k=40000,   l=40001   */
```

Zeile 1 zeigt die Deklaration eines Enums ohne explizite Zuweisung. Dem ersten Element wird in diesem Fall automatisch der Wert 0 zugewiesen. Jedes folgende Element erhält einen um 1 höheren Wert, solange kein anderer Wert bestimmt wird. Als Datentyp für die Elemente des Enums sind in diesem Fall alle Integertypen möglich, da sich die Zahlen 0, 1 und 2 in allen Datentypen speichern lassen.
Für die Elemente von `en2` ist mindestens ein `short` erforderlich, wobei dieses `signed` oder `unsigned` sein kann.
Interessant wird es in Zeile 3, da für die Elemente von `en3` auf jeden Fall ein `signed` Datentyp verwendet werden muss.
Nur im vierten Fall ist der Datentyp anhand der Werte vorgegeben. Er muss die Zahlen -1 und 40001 abdecken. Durch die Zahl -1 muss der Datentyp `signed` sein und die Zahl 40001 kann nur durch ein `long` dargestellt werden. Deshalb hat `en4` den Datentyp `signed long`.[10]
Einige Compiler verwenden für Enumerations generell den Datentyp `unsigned long` und wechseln nur dann auf `signed long`, wenn Elemente explizit negativ deklariert werden. Andere hingegen passen den Datentyp den Werten der Elemente an. Das heißt, sie wählen immer den kleinstmöglichen Datentyp.

[10] Wertebereiche von signed int: [-32768,32767]; signed long: [-2147483648,2147483647] (Zweierkomplement)

3.1.10 Qualifiers

Mit Qualifiers sind `volatile` und `const` gemeint. Implementationsabhängig ist hier, was einen Zugriff auf ein `volatile`-Objekt ausmacht. (What constitutes an access to an object that has volatile-qualified type [ISO90]:6.5.3). Was mit diesem Satz gemeint ist, konnte leider nicht herausgefunden werden. Die eigentliche Bedeutung des `volatile`-Qualifiers wird aber anscheinend dadurch nicht berührt. Deshalb wird auf diesen Punkt nicht weiter eingegangen.

3.1.11 Declarators

C bietet dem Softwareentwickler die Möglichkeit, eigene Typen zu deklarieren oder andere Namen für vorgegeben Datentypen einzuführen. Dabei soll jeder Compiler eine Verschachtelung von mindestens zwölf Namen akzeptieren. Wie viele verschiedene Namen der jeweilige Compiler zulässt, ist implementationsabhängig. Der folgende Beispielcode zeigt eine solche Verschachtelung.

```
1 typedef int level1;
2 typedef level1 level2;
3 typedef level2 level3;
4 int var1;
5 level3 var2;
```

Die Variablen `var1` und `var2` haben in diesem Fall den gleichen Datentyp `int`.
Generell dienen eigene Typdeklarationen dazu, den Code lesbarer machen. Werden sie jedoch zu häufig und verschachtelt angewandt, tritt das Gegenteil ein. Gerade in größeren Projekten ist es wichtig, dass auch Softwareentwickler, die den Code nicht mitentwickelt haben, diesen schnell verstehen und damit arbeiten können. Dabei sind eigene Typennamen jedoch hinderlich, da der neue Softwareentwickler erst wissen muss, was der jeweilige Name zu bedeuten hat und wie dieser Typ verwendet werden darf. Allgemein sind verschachtelte Typdeklarationen bei Bosch nicht üblich, so dass zwölf Schachtelungen ausreichend sind. Nach Rücksprache findet deshalb keine weitere Untersuchung zur Verschachtelung von Deklaratoren statt.

3.1.12 Statements

Der Standard sagt, dass in einem `switch`-Statement mindestens 257 `case`-Werte akzeptiert werden sollen. Ein Compiler kann eventuell mehr verarbeiten, aber diese Menge ist ausreichend, so dass keine weitere Analyse notwendig ist.

3.1.13 Preprocessing directives

Da der Präprozesser auf dem Entwicklungssystem ausgeführt wird, kann es bei bedingten Anweisungen zu anderen Ergebnissen kommen als zur Laufzeit. Dies kann besonders bei Zeichenkonstanten vorkommen, wenn dadurch, dass der auf dem Entwicklungssystem eingesetzte Zeichensatz ein anderer ist als der auf dem Mikroprozessor (siehe Abschnitt 3.1.4) oder wenn die Konstante einen negativen Wert hat ([ISO90]:6.8.1). Das heißt, die folgenden beiden Ausdrücke müssen nicht zum gleichen Ergebnis führen.

```
1 #if 'z' - 'a' == 25          /* Präprozessor Anweisung */
2  if ('z' - 'a' == 25)        /* normaler Code */
```

Da aber auch hier in den meisten Fällen der ASCII-Zeichensatz verwendet wird, genügt dessen Überprüfung und eine Warnung an den Anwender, falls dies nicht der Fall sein sollte.

Der Standard sieht die Präprozessoranweisung `#pragma` für compilerspezifische Befehle und Anweisungen vor. Ob und welche Anweisungen ein Hersteller für seinen Compiler vorsieht, kann allgemein nicht gesagt werden und wird deshalb nicht weiter behandelt. Der Anwender muss in diesem Fall auf das Handbuch des Compilers zurückgreifen.

Auch die restlichen implementationsabhängigen Präprozessoranweisungen sind für diese Analyse nicht weiter interessant oder können nicht geprüft werden, beispielsweise die Methode zum Auffinden von eingebundenen Quelldateien (eingebunden über `#include`) und ob diese auch in Anführungsstrichen stehen können. Hierbei handelt es sich um Eigenschaften des Entwicklungssystems. Es wird vorausgesetzt, dass der Compiler beides vernünftig implementiert hat.

3.1.14 Library functions

Laut dem Standard C90 sind für freistehende Umgebungen (freestanding environments), also Umgebungen, in denen sich das Entwicklungssystem vom ausführenden System unterscheidet, alle Bibliotheksfunktionen implementationsabhängig. Aus diesem Grund wird es bei Bosch weitestgehend vermieden, vordefinierte Bibliotheken zu verwenden. Es würde außerdem den Rahmen dieser Arbeit sprengen, alle im Standard vorgegebenen Funktionen zu überprüfen.

3.2 Zusätzliche Eigenschaften

Neben den „implementation defined features“ gibt es noch andere für den Entwickler interessante Eigenschaften. Diese werden in diesem Abschnitt betrachtet.

3.2.1 Division und Modulo durch Null

Das Ergebnis einer Division oder eines Modulo mit Null ist im Standard undefiniert. Mathematisch ist die Division durch Null nicht definiert und führt bei Computerprogrammen oft zum Absturz. Ein undefinierter Programmabsturz ist in eingebetteten Systemen sehr kritisch, besonders bei sicherheitsrelevanten Anwendungen. Als Alternative wird oft ein definierter Wert zurückgegeben. Dabei sind die folgenden Möglichkeiten üblich:

`a / 0 =`	`b % 0 =`	
`0`	`0`	der Divisior
`a`	`b`	der Dividend
`x`	`y`	eine definierte Konstante, die von beiden unabhängig ist.

Auch wenn versucht wird, eine Division durch Null zu vermeiden, kann dies selten gänzlich ausgeschlossen werden. Deshalb ist es für den Softwareentwickler wichtig zu wissen, was in diesem Fall passiert.

3.2.2 Common Extensions

Wie am Anfgang des Kapitels erwähnt, gibt es im Anhang des Standards C90 auch eine Liste mit allgemein üblichen Erweiterungen zur Programmiersprache. Es handelt sich um neun Punkte (siehe Anhang A), wobei hier nur auf drei eingegangen wird. Dabei handelt es sich um die Unterscheidung von Bezeichnern, anderen arithmetischen Datentypen wie `long long`, und dass Bitfelder nicht nur vom Datentyp `int` sein dürfen.

3.2.3 Makros

Es werden in C oft Makros verwendet, also Präprozessorsymbole, hinter denen sich Code oder auch nur eine Konstante verbirgt. Diese machen den Quellcode lesbarer, wartungsfreundlicher und auch portabler, da bei einer Änderung nur die Definition des Makros geändert werden muss. In den meisten Fällen werden Makros in Headerdateien definiert. Viele Compiler haben aber auch Makros intern vordefiniert. Das Programmsystem soll eine Möglichkeit bieten, den Inhalt solcher Makros anzuzeigen.

3.2.4 Kommentare

Zur Kennzeichnung von Kommentaren ist nach dem Standard C90 nur der mehrzeilige Kommentar mit `/* ... */` zugelassen. In dem neuen Standard C99 wird der einzeilige Kommentar mit `'//'` eingeführt. Da mittlerweile fast alle Compiler den Kommentar mit `'//'` unterstützen, wird dieser auch häufig angewendet. Deshalb soll überprüft werden, ob der Compiler diesen auch akzeptiert.

Kapitel 4

Programmsystem

Das zu entwickelnde Programmsystem wird im Folgenden *CEval* genannt, abgeleitet von Compiler Evaluation.
Bevor jedoch näher auf die Anforderungen und den Aufbau von *CEval* eingegangen wird, beschreibt der folgende Abschnitt den Build-Prozess eines Programms. Dieser hat großen Einfluss auf den Aufbau des Programmsystems.

4.1 Build-Prozess

Der Build-Prozess ist der Weg vom Quellcode zum fertigen Programm. Er umfasst das Kompilieren der verschiedenen Quelldateien in Objekte, das Linken der Objekte und notwendigen Bibliotheken[1] und die Erstellung von ausführbarem Maschinencode. Bei größeren Projekten mit mehreren Quelldateien und Bibliotheken werden normalerweise Skripte (Perl, Batch usw.) eingesetzt, um so die vielen Kompiliervorgänge, das Linken und andere nötige Vorgänge zu automatisieren. Oft wird auch das Tool Make eingesetzt. Dieses Tool hat den Vorteil, dass einmal kompilierte Quelldateien nur dann neu kompiliert werden, wenn in ihnen etwas geändert wurde. Bei großen Projekten mit mehreren hundert Quelldateien kann so der Build-Prozess bei kleinen Änderungen deutlich verkürzt werden.

Sei es nun ein Skript oder Make, der Build-Prozess läuft ähnlich ab und es müssen ähnliche Voreinstellungen gemacht werden. Normalerweise werden neben den benötigten Quelldateien auch die verwendeten Bibliotheken angegeben. Da Compiler und Linker üblicherweise getrennt aufgerufen werden, muss angegeben werden, wie und mit welchen Einstellungen diese gestartet werden sollen. Obwohl der Linker im Prinzip schon fertigen Maschinencode liefert, wird dieser oft noch weiterverarbeitet. Zum einen, um beispielsweise eine Verifizierung durch Checksummen hinzuzufügen oder um das Herunterladen des Maschinencodes

[1] Sammlung schon kompilierter Quellen

auf den Mikroprozessor bzw. das Steuergerät zu vereinfachen. Deshalb werden nach dem Linken oft noch andere Programme aufgerufen, die dann zum Beispiel eine `elf`[2]-Datei erstellen. Diese wird zum Schluss auf die Hardware heruntergeladen.

Der genaue Aufbau des Build-Prozesses und die jeweiligen Einstellungen unterscheiden sich je nach Projekt zum Teil deutlich, sogar dann, wenn der gleiche Compiler und der gleiche Mikroprozessor zum Einsatz kommen sollte.

4.2 Anforderungen an das Programmsystem *CEval*

Aus der Aufgabenstellung in 1.2 geht hervor, dass ein möglichst einfach zu bedienendes Programmsystem zu entwickeln ist, welches bestimmte Eigenschaften von Compilern abprüft. Die zu prüfenden Eigenschaften ergeben sich zum großen Teil aus den „implementation defined features“ des Standards C90, wobei das Augenmerk auf denen liegt, die für eingebettete Systeme relevant sind. Dies soll unter den gleichen Bedingungen geschehen, unter denen der Compiler auch im jeweiligen Projekt eingesetzt wird, da einige der untersuchten Eigenschaften von den Einstellungen des Compilers abhängen. Weiterhin muss das Programm flexibel sein, um in Zukunft auch neue Compiler testen zu können. Im Gesamten soll die Analyse weitestgehend automatisch ablaufen.

Neben diesen von Bosch vorgegebenen Anforderungen wird der Aufbau des Programmsystems auch stark von den durchzuführenden Tests bestimmt (siehe Kapitel 3).
Es stellt sich die Frage, welche Schritte für eine umfassende Analyse überhaupt notwendig sind. Die implementationsabhängigen Eigenschaften können in drei Klassen unterteilt werden (siehe Kapitel 3).

- Nicht prüfbare Eigenschaften wie Diagnosemeldungen.
  Hier sind dem Hersteller so viele Freiheiten gegeben, dass nicht alle Möglichkeiten abgesehen und deshalb auch nicht sinnvoll getestet werden können. Weiterhin können einige speicherbezogene Eigenschaften nur überprüft werden, wenn der Assemblercode oder die `map`-Datei analysiert werden. Darauf ist das Programmsystem nicht ausgelegt, da sich auch diese von Compiler zu Compiler stark unterscheiden können.

- Eigenschaften, die sich nur beim Kompilieren oder Linken feststellen lassen, wie die maximale Länge von Bezeichnern oder, ob der Compiler eine Division durch Null erkennt.
  Hier müssten Compilermeldungen analysiert werden, was aber im ersten Punkt ausgeschlossen wird. Eine andere Möglichkeit wird in Abschnitt 4.4.2 erklärt.

[2]elf: executable and linking format. Es gibt auch noch diverse andere Formate. Die einfachste Form wäre der reine Maschinencode in einer hex-Datei

- Eigenschaften, die in einem laufenden Programm überprüft werden.
  Dies sind die meisten Eigenschaften, zum Beispiel wie Datentypen umgewandelt werden, negative Zahlen intern dargestellt werden usw.

Für die Tests ergeben sich somit zwei Hauptphasen, das Kompilieren und Linken, zusammengefasst im Build-Prozess, sowie der tatsächliche Programmlauf auf dem Mikroprozessor. Diese beiden Prozesse werden für die verschiedenen Tests mehrmals durchlaufen.
Zum Schluss müssen dann noch die Ergebnisse aus den verschiedenen Durchläufen zusammengefasst, ausgewertet und in eine leserliche Form gebracht werden. Dadurch kommt ein abschließender Prozess hinzu. Es ergeben sich die folgenden drei Prozesse, die unterschiedlich oft durchlaufen werden.

1. Kompilieren des Quellcodes und Linken der verschiedenen Objekte (Build-Prozess)
2. Test auf dem Mikroprozessor (Programmlauf)
3. Auswertung und anwenderfreundliche Darstellung der Ergebnisse (Auswertung)

4.3 Datenmodell

Bei der Entwicklung von Software für eingebettete Systeme werden, wie in Abschnitt 2.1 beschrieben, Crosscompiler eingesetzt. Der Code für den Mikroprozessor wird auf einem PC oder anderem System generiert und dann auf den Mikroprozessor geladen. Die Ergebnisse des Programmlaufs müssen der Auswertung zur Verfügung gestellt werden, das heißt, es müssen auch Daten vom Mikroprozessor gelesen werden. Das Laden und Auslesen von Daten auf und vom Mikroprozessor lässt sich während der Entwicklungsphase nur schwer automatisieren. Deshalb müssen die drei gerade definierten Prozesse getrennt betrachtet werden. Das Kompilieren und Linken sowie die Auswertung finden auf dem PC statt, wohingegen der Programmlauf auf dem Mikroprozessor durchgeführt wird. Das Datenflussdiagramm in Abbildung 4.1 stellt diesen Vorgang graphisch dar.

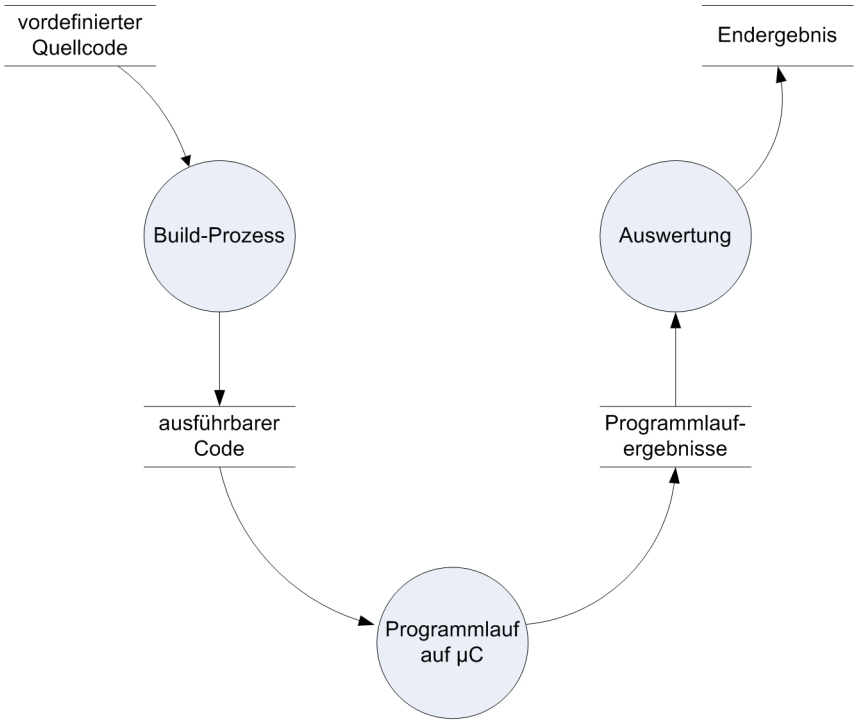


Abbildung 4.1: Datenflussdiagramm - einfach

Wie in Abschnitt 4.2 beschrieben, werden schon beim Kompilieren Eigenschaften untersucht. Dies bedeutet, dass auch diese Ergebnisse in die Auswertung einfließen müssen. Weiterhin werden einige Tests auf den Ergebnissen von anderen Tests aufbauen. Diese müssen wieder in den ersten Prozess zurückfließen. Das heißt auch, dass nicht der komplette Quellcode vordefiniert werden kann. Vor dem Kompilieren und Linken wird dementsprechend ein Prozess benötigt, der auf Basis des vordefinierten Quellcodes und der vorherigen Ergebnisse den Quellcode für den aktuellen Durchlauf generiert. Daraus ergibt sich das in Abbildung 4.2 dargestellte Diagramm.

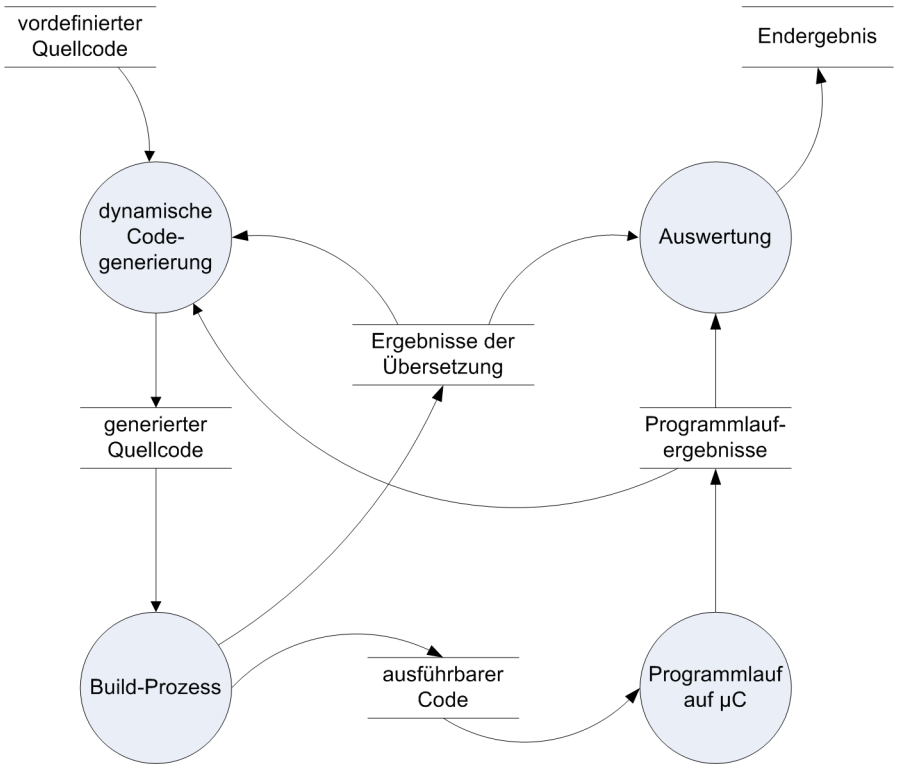


Abbildung 4.2: Datenflussdiagramm - erweitert

4.4 Lösungsansätze

Aus den Anforderungen und dem Datenmodell ergeben sich mehrere Fragen und Schwierigkeiten:

- Wie wird die nötige Flexibilität bezüglich des zu analysierenden Compilers und der Umgebung, in der er eingesetzt werden soll, erreicht?
- Wie können Compilerfehler ausgewertet werden?

- Wie kommt der Maschinencode auf den Mikroprozessor?
- Wie werden die Ergebnisse der Laufzeittests vom Mikroprozessor gelesen?
- Wie werden Zwischenergebnisse abgespeichert?
- Wie kann Quellcode auf Basis von früheren Ergebnissen dynamisch generiert werden?
- Das Programm soll erweiterbar sein.
- Das Programm soll einfach und ohne viel Aufwand bedienbar sein.

Im Folgenden wird auf diese Punkte näher eingegangen und es werden verschiedene Lösungsansätze miteinander verglichen. Dabei gilt das Hauptaugenmerk der geforderten Flexibilität, um ein möglichst breites Spektrum an C-Compilern untersuchen zu können.

4.4.1 Flexibilität

Wie kann das Programm so flexibel gestaltet werden, dass auch neue Compiler damit analysiert werden können und die Analyse möglichst projektspezifisch ist?

Der erste Ansatz zur Lösung dieser Frage war die Nutzung einer Konfigurationsdatei. Hier muss der Anwender die Daten für den jeweiligen Compiler eintragen. Das Programm verwendet dann diese Einstellungen, um den Compiler aufzurufen und die Analysen durchzuführen. Das Problem dieses Ansatzes ist, dass er wenig mit dem aktuellen Projekt des Anwenders zu tun hat. Der Anwender kann zwar die gleichen Compilereinstellungen angeben, es kann jedoch sein, dass neben dem Compiler und dem Linker noch andere Tools zum Einsatz kommen, die das Ergebnis abändern können wie in Abschnitt 4.1 erklärt. Aus diesen Gründen musste eine andere Lösung gefunden werden.

Der zweite Ansatz ergibt sich aus der Aufgabenstellung. Da diese Analyse jeweils für ein bestimmtes Projekt durchgeführt werden soll, ist es sinnvoll, dass der Anwender den Build-Prozess zur Verfügung stellt. So ist die Nähe zum eigentlichen Projekt garantiert und auch die Flexibilität, die nötig ist, um zukünftige Compiler zu analysieren. Der Anwender kann dafür seinen schon bestehenden Build-Prozess nehmen, einige Änderungen vornehmen und dem Programm mitteilen, wie der Prozess aufgerufen wird. Auf die Änderungen wird in Abschnitt 5.3.1 eingegangen.

Neben dem Build-Prozess soll sich auch die Software zur Analyse der Eigenschaften mühelos in bestehende Projektumgebungen integrieren lassen.
Die Idee dabei ist, dass der Anwender nur eine zusätzliche Funktion aufrufen muss, um die relevanten Tests zu starten. Hierbei muss er aber darauf achten, dass ungewollte Nebeneffekte, zum Beispiel durch einen Watchdog, vermieden werden.

4.4.2 Compilerfehler

Compilerfehler treten auf, wenn der Compiler während des Kompilierens Fehler im Quellcode entdeckt. Laut Standard C90 muss dann eine Diagnosemeldung angezeigt werden und der Vorgang bricht ab. Wie schon erwähnt, sind dem Compilerhersteller beim Verfassen dieser Meldungen vom Standard her große Freiheiten gegeben, so dass sich diese Meldungen nicht automatisch analysieren lassen.
Eine Möglichkeit wäre, den Anwender die Compilermeldungen analysieren zu lassen und diese dann in einer definierten Form zu erfassen und weiterzuverarbeiten. Dafür müsste sich der Anwender aber mit dem Compiler auskennen und es würde das Arbeiten mit dem Programm wesentlich erschweren.
Stattdessen wird davon ausgegangen, dass die zu analysierenden Compiler konform zum Standard C90 sind. So kann Code geschrieben werden, der laut Standard kompilierbar sein muss. Implementationsabhängige Eigenschaften, die sich nur während des Kompilierens prüfen lassen, können dann so in den Code eingebaut werden, dass sie je nach Implementierung kompiliert werden können oder einen Fehler verursachen. Das heißt, es werden kleine Module erstellt, in denen erwartungsgemäß nur an exakt einer Stelle ein Compilerfehler auftreten kann. Diese Module werden dann einzeln kompiliert. Anhand des Erfolgs oder Misserfolgs des Kompilierens lässt sich dann die Implementierung der jeweiligen Eigenschaft erkennen, ohne die Diagnosemeldungen des Compilers zu analysieren.
Dieses Vorgehen erhöht zwar die Anzahl der vorzunehmenden Durchläufe erheblich, lässt sich dafür aber automatisieren, was die Bedienbarkeit erhöht. Da der Vorgang im Hintergrund ablaufen kann, ist es nicht wichtig, ob es zehn Minuten oder drei Stunden dauert. Der Anwender kann währenddessen andere Arbeiten erledigen.
Damit dieses System funktioniert, muss dem Programmsystem *CEval* mitgeteilt werden, ob der Kompiliervorgang erfolgreich war. Um von Rückgabewerten unabhängig zu sein, mit denen ein Build-Prozess eventuell schon seinen Status signalisiert, wird dem Anwender vorgegeben, dass bei einem erfolgreichen Durchlauf eine bestimmte Textdatei erzeugt werden muss. Das Programmsystem prüft, ob diese Textdatei vorhanden ist. Falls dies nicht der Fall ist, trat beim Build-Prozess ein Fehler auf.

4.4.3 Programmierung des Mikroprozessors

Ein weiterer Arbeitsschritt, der sich nicht automatisieren lässt, ist das Herunterladen des Maschinencodes auf den Mikroprozessor. Der Anwender muss dafür sorgen, dass der ausführbare Maschinencode übertragen und der Programmlauf gestartet wird. Damit das Herunterladen so selten wie möglich geschehen muss, werden die Tests, im Gegensatz zur Vorgehensweise für Compilerfehler, so weit wie möglich zusammengefasst und anhand vorheriger Ergebnisse angepasst. So soll sichergestellt werden, dass der Code, der tatsächlich

auf dem Mikroprozessor läuft, keine Compilerfehler erzeugt.
Es lässt sich jedoch nicht vermeiden, auch auf dem Mikroprozessor mehrere Durchläufe auszuführen. Einige Analysen basieren auf Ergebnissen von vorherigen Analysen. Es gibt auch Analysen, die zu einem Laufzeitfehler führen können, wie beispielsweise die Untersuchung des Verhaltens bei einer Division durch Null.

4.4.4 Datenübertragung

Neben der Programmierung des Mikroprozessors kann auch das Lesen der Ergebnisse des Programmlaufs von dem Mikroprozessor nicht vorher festgelegt werden. Da sich die Umgebung, in der der Mikroprozessor eingesetzt wird, von Projekt zu Projekt unterscheidet, kann dieser Punkt nicht allgemein gelöst werden. Wichtig für eine automatische Weiterverarbeitung der Ergebnisse ist ein festgelegtes Format und eine definierte Stelle (Datei), an der die Daten abgelegt werden.
Wie nun genau die Daten übertragen werden, ist vom Anwender festzulegen. Die Idee ist, dass er dazu eine C-Funktion schreibt. Diese Funktion wird von *CEval* eingebunden und dazu benutzt, die Ergebnisse auszugeben.
Bei den Tests während der Entwicklung wurde dazu die serielle Schnittstelle benutzt, so dass die Daten direkt zum PC übertragen und dort aufgezeichnet wurden. Denkbar sind aber auch andere Wege, wie CAN[3]-Nachrichten, Zwischenspeichern der Daten in einem EEPROM oder das manuelle Auslesen der Daten über eine Debugumgebung.
Auch wenn dieser Weg kompliziert ist und die Bedienbarkeit des Programms erschwert, gibt es keine andere Möglichkeit, die die nötige Flexibilität für projektspezifische Lösungen bietet.

4.4.5 Kommunikation zwischen den Prozessen

Die Evaluierung findet in mehreren Schritten und normalerweise auf zwei verschiedenen Systemen statt. Wie gestaltet man am Besten die Kommunikation zwischen den Prozessen?
Da der Programmablauf normalerweise manuell gestartet werden muss, können diese Schritte nur bedingt automatisiert werden. Daten können also nicht direkt übergeben werden. Deshalb ist es am sinnvollsten, diese in temporären Dateien auf der Festplatte abzulegen. Dabei muss gewährleistet werden, dass alle Prozesse auf die gleichen Daten zugreifen. Weiterhin muss das Format festgelegt werden, in dem die Daten abgespeichert werden. Die einfachste Lösung sind in diesem Fall Textdateien. Die Daten können im Klartext gespeichert werden. Eine Alternative wäre das XML-Format, das aber zu diesem

[3]Controller Area Network, ein von Bosch entwickeltes Bussystem

Zweck überdimensioniert ist. Außerdem müsste dafür zusätzlich noch ein XML-Parser implementiert werden.

4.4.6 Dynamische Generierung von Quellcode

Ein wichtiger Punkt ist das Zusammenstellen des Quellcodes für die verschiedenen Tests. Die Analyse des Standards zeigt, dass einige Eigenschaften während der Kompilier- und Linkphase überprüft werden müssen und zu Fehlern führen können. Besonders die Suche nach der maximalen Länge von Bezeichnern ist ein iterativer Prozess, der mehrere Kompiliervorgänge benötigt. Andererseits können für die meisten Eigenschaften Funktionen geschrieben werden, die die jeweilige Eigenschaft abprüfen. Es gibt nun drei Möglichkeiten:

1. Der Quellcode wird komplett dynamisch generiert. Dadurch wird das Programm allerdings unübersichtlich und läßt sich nicht sehr gut warten.

2. Wird der Quellcode komplett vordefiniert und je nach Situation der entsprechende Code ausgewählt, müssen sehr viele Möglichkeiten erfasst werden. Das heißt, die Menge an vordefiniertem Quellcode, der sich zum Teil nur geringfügig voneinander unterscheidet, wird unüberschaubar.

3. Die sinnvolle Alternative ist eine Kombination vorgenannten Möglichkeiten. Wie gesagt, lassen sich die meisten Funktionen vordefinieren. Unterschiede lassen sich zum Teil über Präprozessordirektiven einstellen, indem z.B. Headerdateien umdefiniert werden. Für andere Eigenschaften können auch Codefragmente während des Durchlaufs generiert werden. Das folgende Beispiel zeigt, wie eine Auswahl über Makrodefinitionen aussehen kann.

   ```
   1 #ifdef CEVAL_MAKRO
   2   printf("CEVAL_MAKRO ist gesetzt");
   3 #endif
   ```

   Der Präprozessor überprüft, ob das Makro `CEVAL_MAKRO` gesetzt ist. Falls nicht, wird der Code in Zeile 2 ignoriert.

4.4.7 Erweiterbarkeit

Das zu entwickelnde Programmsystem analysiert nur die in Kapitel 3 beschriebenen Eigenschaften. Zum Teil werden Eigenschaften ausgelassen, weil sie irrelevant für eingebettete Systeme sind, nach Rücksprache nicht notwendig sind oder aber ihre Implementierung zu

aufwendig ist. Ebenso können in Zukunft noch weitere Eigenschaften hinzukommen, die untersucht werden sollen.
Es ist also wichtig, die Erweiterbarkeit sicherzustellen. Dazu sind zwei Dinge notwendig. Erstens ein offener, verständlicher und gut erweiterbarer Aufbau des Programmsystems und zweitens eine gute Dokumentation des Ganzen.
Der Aufbau wird zum großen Teil durch die Anforderungen und dem Umfeld aus zwei verschiedenen Systemen vorgegeben. Hier wird, wie in den vorherigen Abschnitten beschrieben, der Anwender mit eingebunden. Er sichert durch den eigenen Build-Prozess und die zu implementierende C-Funktion die nötige Flexibilität, neue Compiler zu analysieren.
Eine gute Dokumentation ist für das Verständnis wichtig. Neben einem Handbuch für den Anwender und diesem Dokument ist eine ausreichende Dokumentation des Quellcodes notwendig. Außerdem ist ein „Administrator"-Handbuch sinnvoll. Für C bietet sich das freie Tool Doxygen[4] an, mit dem sich umfangreiche Dokumentationen direkt aus dem Quellcode generieren lassen.

4.4.8 Aufwand für den Anwender

Voraussetzung für die Nutzung des Programmsystems ist eine gewisse Erfahrung in der Programmiersprache C und der Programmierung von Mikroprozessoren. Die Priorität liegt auf der Flexibilität, deshalb lässt sich der Aufwand für den Anwender nur bedingt minimieren. Die verschiedenen Lösungsansätze zeigen, dass die Evaluierung nicht ohne die Mithilfe des Anwenders laufen kann. Die folgende Auflistung soll nochmals zeigen, welche Aufgaben vom Anwender übernommen werden müssen:

- Bereitstellung eines angepassten, projektspezifischen Build-Prozesses.
- Programmierung einer Funktion zur Übermittlung von Daten vom Mikroprozessor zum Entwicklungssystem.
- Übertragen des Maschinencodes auf den Mikroprozessor, Starten des Programmlaufs und Übergabe der Ergebnisse an die Auswertung.

[4]Source code documentation generator tool; Dimitri van Heesch; `http://www.doxygen.org`

Kapitel 5

Implementierung

In diesem Kapitel werden die Algorithmen und Analysen für die verschiedenen Eigenschaften aus Kapitel 3 sowie die Umsetzung der einzelnen Lösungsansätze aus Kapitel 4 beschrieben.

5.1 Allgemeines

5.1.1 Programmiersprache

Obwohl es um die Programmiersprache C geht, entstand die Frage, in welcher Sprache das Programmsystem implementiert werden sollte. Für die Tests der einzelnen Eigenschaften muss C verwendet werden. Der Programmablauf kann aber in jeder beliebigen Sprache implementiert werden. Hierbei war es wichtig, dass das Programm ohne Probleme von den Boschmitarbeitern genutzt und gepflegt werden kann.
Als erste Option kam C in Frage. Für C spricht, dass insgesamt nur eine Sprache verwendet wird. Weiterhin bietet sich dadurch die Möglichkeit, für das Programm und die Tests auf die gleichen Quellen zurückzugreifen. Es werden zum Beispiel viele Ergebnisse der einzelnen Tests über Integerkonstanten zurückgegeben (siehe 5.2.1.2). Diese können in einem Header definiert und global verwendet werden. Der Nachteil von C ist die komplizierte Stringverarbeitung und die schwierige Interaktion mit dem System.
Weitere Optionen waren Java, Python und Perl. Nach Absprache mit den Kollegen, die das Tool verwenden sollen, fiel die Entscheidung auf die Programmiersprache Perl. Perl hat seine Stärken in der Bearbeitung von Textdateien und der Steuerung von anderen Programmen. Es ist eine Skriptsprache, mit der sich einfach und schnell wiederkehrende Prozesse automatisieren lassen. Die meisten Entwickler bei Bosch haben Erfahrung im Umgang mit Perl und es ist dort auf jedem Entwicklungs-PC vorinstalliert.

5.1.2 Entwicklungsumgebung

Die Implementierung der verschiedenen Programmteile fand auf einem PC mit einem Intel Pentium 4 Prozessor statt. Die Tests der einzelnen Funktionen für die Analyse fanden auf dem PC mit dem GNU C-Compiler Version 4.2 statt. Als zweites System stand ein M68HC12 Mikroprozessor der Firma Freescale[1] in Verbindung mit dem Cosmic HC12 C-Compiler Version 4.6f zur Verfügung. Auf diesem System wurde zum großen Teil auch getestet. Zum Debuggen wurde ein „On-Chip-Debugger“ der Firma iSYSTEM in Verbindung mit der Entwicklungsumgebung „winIDEA“ genutzt.[2]
Neben diesen beiden Systemen wurden bei späteren Tests auch andere Compiler und Prozessoren eingesetzt (siehe Kapitel 6).
Das Programmsystem wurde mit dem Perl Interpreter Version 5.8.4 von ActiveState[3] entwickelt und getestet.

5.2 Analysen

Die nun folgenden Abschnitte befassen sich mit verschiedenen Themen, die bei der Implementierung des C-Codes für den Programmlauf beachtet werden müssen, sowie den eigentlichen Analysen der in Kapitel 3 genannten Eigenschaften.

5.2.1 Allgemein

Da der generierte Code mit vielen verschiedenen Compiler-Prozessor-Kombinationen funktionieren soll, muss er portabel sein. Das heißt, der Code, der nicht direkt mit einer Analyse zu tun hat, darf keine Funktionen, Konstrukte oder ähnliches enthalten, die nicht genau im Standard definiert sind. Aus diesem Grund wird auch auf die Verwendung von Bibliotheksfunktionen verzichtet (vgl. Abschnitt 3.1.14).
Daraus ergibt sich für die zu analysierenden Compiler die Voraussetzung, dass sie nach dem Standard C90 implementiert sein müssen. Sonst kann die Funktionalität der entwickelten und implementierten Algorithmen nicht gewährleistet werden.

5.2.1.1 Namenskonvention

Die Analysen sollen so nah wie möglich am eigentlichen Projekt des Anwenders stattfinden. Deshalb muss beachtet werden, dass sich keine Bezeichner des von *CEval* erzeugten C-

[1] `http://www.freescale.com`
[2] `http://www.isystem.com`
[3] `http://www.ActiveState.com`

Codes mit Bezeichner des anderen C-Codes überschneiden. Deshalb bekommt jede globale Funktion, jede Variable und jedes Präprozessorsymbol das Präfix

`RB_CEVAL` für Makros
`rbCEval` für alle anderen globalen Bezeichner

Dabei ist `rb` ein allgemeines Präfix für die Robert Bosch GmbH und `CEval` verweist auf dieses Programmsystem.
Um zusätzlich zur Lesbarkeit des Codes beizutragen, wird jedem Variablennamen ein Suffix angehängt, der den Datentyp erkennen lässt. So bezeichnet beispielsweise `counter_psi` einen Pointer auf eine `signed integer`-Variable mit dem Namen `counter`, und `var_c` eine `char`-Variable.

5.2.1.2 Arten von Analysen

Es gibt zwei Arten von Analysen. Bei der einen wird ein Wert berechnet, der auch das Ergebnis darstellt, wie zum Beispiel die Anzahl der Bits in einem Byte oder die Größe der Mantisse von einem `float`. Bei den meisten Analysen wird jedoch ein bestimmtes Verhalten geprüft und das Ergebnis ist auf eine bestimmte Anzahl von Aussagen begrenzt. In diesem Fall wird für jedes mögliche Ergebnis ein konstanter Wert festgelegt, dem in der Auswertung dann der Aussage zugeordnet wird. Ein einfaches Beispiel hierzu ist das Rundungsverhalten bei einer negativen Integerdivision. Das kann laut Standard nur gegen Null oder gegen minus unendlich sein, es gibt also zwei Möglichkeiten. Wird noch davon ausgegangen, dass das Rundungsverhalten eventuell nicht bestimmt werden kann, gibt es noch eine dritte Möglichkeit. Diese drei Konstanten werden im Header `rbCEvalConst.h` vordefiniert und lauten in diesem Fall:

```
  /* error */
#define RB_CEVAL_DIV_NEG_ERROR        9
  /* abgeschnitten (zu Null gerundet) */
#define RB_CEVAL_DIV_NEG_TRUNC        1
  /* kleiner oder gleich (zu minus unendlich gerundet */
#define RB_CEVAL_DIV_NEG_LESS_EQUAL 2
```

Sprechende Namen für die Konstanten sorgen somit für besser lesbaren Quellcode. Allgemein haben die Funktionen für die Analysen den folgenden Aufbau:

```
void rbCEval_Funktionsname(
                Parameter_1,
                Parameter_2,
                ...,
```

```
Zeiger auf Ergebnis_1,
...)
```

Das heißt, Ergebnisse werden einheitlich nicht als Rückgabewert ausgegeben. Stattdessen muss die Adresse, an der das Ergebnis gespeichert werden soll, an die Funktion übergeben werden. Das hat den Vorteil, dass Funktionen, die nur ein Ergebnis liefern und Funktionen, die mehrere Ergebnisse liefern, den gleichen Aufbau haben.

Neben den Funktionen für die eigentlichen Tests gibt es noch Funktionen zur allgemeinen Verwendung.

5.2.1.3 Ausgabefunktionen

Die Ausgabe vom Mikroprozessor zurück zum Entwicklungsrechner findet, wie in Abschnitt 5.3.2 beschrieben, zeichenweise statt und muss vom Anwender in der Funktion `rbCEval_outputByte(unsinged char)` bereitgestellt werden. Zeichenweise heißt hier, dass für jedes Zeichen der zugehörige ASCII-Code übertragen wird. Da der Zeichensatz auf dem Mikroprozessor aber nicht ASCII sein muss, wird jedes Zeichen vor der Übertragung in den entsprechenden ASCII-Code umgewandelt. Dafür sorgen interne Ausgabefunktionen, die auf die bereitgestellte Funktion `rbCEval_outputByte(unsinged char)` zurückgreifen. Intern gibt es die folgenden Ausgabefunktionen für Zahlen, Zeichenketten (Strings) und einzelne Zeichen.

```
void rbCEval_outputNumber(long);
void rbCEval_outputString(char*);
void rbCEval_outputChar(char);
```

Zahlen werden dementsprechend auch als String ausgegeben, müssen aber zuerst umgewandelt werden. Dies ginge mit der Bibliotheksfunktion `sprintf()`. Da jedoch auf Bibliotheken verzichtet werden muss, wurde die Umwandlung neu implementiert. Es können daher nur ganze Zahlen des Wertebereichs von `long`[4] ausgegeben werden.

Damit die Ergebnisse für die Auswertung lesbar sind, werden sie im Format `name=wert` ausgegeben. Für die Auswertung muss dann nur bekannt sein, welche Eigenschaft sich hinter dem Namen verbirgt und welche Bedeutung der Wert hat.
Zur vereinfachten Ausgabe der Ergebnisse gibt es die folgende Ausgabefunktion, die direkt das geforderte Format erstellt und anhand derer zu sehen ist, wie die anderen drei Funktionen eingesetzt werden.

[4] $(-2^{31}, 2^{31})$

```
1 void rbCEval_outputResult(char* name_pc, int result_i)
2 {
3   rbCEval_outputString(name_pc);
4   rbCEval_outputChar('=');
5   rbCEval_outputNum(result_i);
6   rbCEval_outputChar('\n');
7 }
```

5.2.1.4 Weitere Hilfsfunktionen

Benötigte mathematische Operationen, wie beispielsweise Potenzen und Betrag, sind in Hilfsfunktionen implementiert. So wird vermieden, dass entsprechende Bibliotheken eingebunden werden müssen. Auf diese Funktionen wird nicht näher eingegangen, da sie nur indirekt mit den zu implementierenden Analysen zusammenhängen.

5.2.1.5 Headerdateien

Alle Funktionen und Konstanten sind in den folgenden vier Headerdateien definiert.

- `rbCEvalOut.h`
  Hier sind die Funktionsprototypen für die Ausgabefunktionen hinterlegt. Um die Wartbarkeit des Programmsystems zu vereinfachen, kann in dieser Datei das Makro `RB_CEVAL_DEBUG` definiert werden. Damit werden Debugmeldungen aktiviert, die an passenden Stellen im Code eingefügt wurden.

- `rbCEval.h`
  In diesem Header sind alle Funktionen definiert, die für die Analysen verwendet werden, ebenso Datenstrukturen und Makros, die im Code verwendet werden. Weiterhin werden in dieser Datei auch die nächsten beiden Header eingebunden.

- `rbCEvalConst.h`
  Diese Datei beinhaltet alle vordefinierten Integerkonstanten, die als Ergebnis zurückgegeben werden, wenn es nur eine begrenzte Anzahl von Ergebnissen für eine Analyse gibt (siehe Abschnitt 5.2.1.2). Damit diese Konstanten auch von einem Perlskript für die Auswertung genutzt werden können, werden sie in einer csv-Datei verwaltet, aus der dieser Header erzeugt wird (siehe Abschnitt 5.3.3.2).

- `rbCEvalPredef.h`
  In diesem Header werden anhand der Ergebnisse aus den Kompiliervorgängen Makros definiert, mit denen nach dem Prinzip aus Abschnitt 4.4.6 Codefragmente aktiviert oder deaktiviert werden können.

5.2.2 Algorithmen zur Analyse der Eigenschaften

In Kapitel 3 werden die zu analysierenden Eigenschaften untersucht und näher beschrieben. In diesem Abschnitt wird die Implementierung zur Überprüfung der Eigenschaften erklärt. Dabei wird nicht der komplette C-Code aufgeführt, sondern nur der Bereich, der für die Erklärung notwendig und hilfreich ist. Zum Teil werden auch Alternativen aufgezeigt und miteinander verglichen. Die Tests sind thematisch zusammengefasst. Am Anfang jedes Unterpunkts wird auf die zugehörigen Abschnitte in Kapitel 3 verwiesen.

5.2.2.1 Größe eines Byte

Referenz: 3.1.4
Ein Byte wird in C mit dem Datentyp `char` repräsentiert. Es gilt herauszufinden, mit wie vielen Bits ein `char` intern dargestellt wird.
Eine Möglichkeit ist das parallele Hochzählen zweier Variablen unterschiedlichen Typs, bis ihr Wert nicht mehr gleich ist.

```
1 unsigned char var_uc = 0;
2 unsigned long var_ul = 0;
3 do{
4    var_uc++;
5    var_ul++;
6 } while (var_uc == var_ul)
```

Nach Beendigung der `while`-Schleife kann dann anhand der `long`-Variable `var_l` die Anzahl der Bits, die ein `char` benötigt, berechnet werden. Solange der Test auf Mikroprozessoren mit einer Bytegröße von 8 Bit läuft, kommt es zu 256 Wiederholungen. Sollte der Test aber auf einem Prozessor mit einer höheren Bytegröße laufen, erhöht sich die Anzahl der Wiederholungen exponentiell mit jedem zusätzlichen Bit.
Eine bessere Variante bietet der Shift Operator. Hierbei werden die beiden Variablen nicht hochgezählt, sondern es wird ein Bit solange verschoben, bis es beim `char` „herausfällt“:

```
1 unsigned char var_uc = 1;
2 bits_i = 0;
3 do
4 {
5    var_uc <<= 1;
6    bits_i++;
7 } while (var_uc != 0);
```

Bei diesem Test werden nur so viele Wiederholungen benötigt, wie Bits in einem Byte sind, es sind keine weiteren Berechnungen nötig.

5.2.2.2 Darstellung negativer Zahlen

Referenz 3.1.5
Negative Integerzahlen werden üblicherweise im Zweierkomplement dargestellt. Diese Darstellung wird auch für die anderen Tests vorausgesetzt. Trotzdem wird überprüft, ob dies auch wirklich der Fall ist. Zu diesem Zweck wird ein vorgegebenes Bitmuster erstellt und dann überprüft, welche Zahl dadurch dargestellt wird. Eine `signed char`-Variable wird dazu bis auf das LSB mit Einsen gefüllt ($= ...11110_2$). Mit diesem Muster wird eine negative Null im Einerkomplement vermieden. Das Bitmuster hat je nach interner Darstellung den folgenden Wert:

-2	Zweierkomplement
-1	Einerkomplement
$-(2^{Bits-1}-2)$	MSB ist Vorzeichenbit

5.2.2.3 Datentypen

Referenz 3.1.4, 3.1.5, 3.1.6, 3.1.9
Alle arithmetischen Datentypen haben festgelegte Mindestgrößen. Zur Bestimmung der tatsächlichen Größe kann der Operator `sizeof` verwendet werden. Die Größe jedes Standarddatentypen ist dazu im Compiler hinterlegt, damit für den jeweiligen Datentyp entsprechend dessen Größe Speicher reserviert werden kann. Die Standarddatentypen sind:

`char, short, int, long, float, double, long double`

Für die Konfiguration von QA-C interessieren zusätzlich die folgenden Größen:

Rückgabewert von `sizeof` (entspricht dem Datentypen `size_t`)
„wide character“
Datenpointer
Codepointer
Differenz von Pointern
`long long`
`void` *(Wird für QA-C nicht benötigt, nur interessehalber untersucht.)*

Da `void` aber kein „complete type“ ist und `long long` im C90 noch kein Standarddatentyp ist, muss vorher überprüft werden, ob deren Größe überhaupt definiert ist. Diese Überprüfung findet separat statt, da ein `sizeof(void)` oder `sizeof(long long)` zu einem Compilerfehler führen kann. Kommt es zu einem Fehler, wird für den entsprechenden Typen die Überprüfung durch Makros ausgeschaltet (siehe Abschnitt 4.4.6), die Größe auf Null gesetzt und dann während der Auswertung als „not defined“ interpretiert.
Da sich die Größe einer Variablen vom Datentyp `enum` sogar je nach Deklaration unterscheiden kann, sind umfangreichere Tests nötig, die gesondert im nächsten Abschnitt

erklärt werden.
Bei dem Rückgabewert von `sizeof` und bei `enum` handelt es sich intern nicht um eigene Datentypen. Anhand deren Größe kann aber festgestellt werden, mit welchem der Standarddatentypen diese Werte intern dargestellt werden. QA-C nutzt diese Informationen, um den Nutzer zu warnen, wenn im Code andere Datentypen angenommen werden und es dadurch zu falschen Ergebnissen kommen kann.

Neben der Größe spielt auch das Alignment, also die Ausrichtung im Speicher eine Rolle. Tests mit verschiedenen Compilern haben gezeigt, dass das Alignment sehr unterschiedlich sein kann. Es wird für die folgenden Typen überprüft (wenn diese definiert sind):

`char, short, int, long, long long, float, double, long double`
Datenpointer, Codepointer

Das Alignment kann, wie in Abschnitt 3.1.9 erklärt, mit Hilfe von Strukturen herausgefunden werden. Wenn ein `char` mit dem zu untersuchenden Datentyp kombiniert wird, gibt die Differenz der beiden Adressen dessen Alignment an. Wegen den Eigenheiten der Pointerarithmetik können die Adressen verschiedener Datentypen nicht direkt verglichen werden. Deshalb wird der zu untersuchende Datentyp (in diesem Fall `long`) in einer Union mit einem `char` kombiniert. Laut Standard C90 sind die Adressen der Elemente einer Union immer gleich.

```
1 struct{
2   char var_c;
3   union{
4     long var_l;
5     char help_c;
6   }un;
7 } long_str;
8 /* Differenz von den Adressen der beiden char Variablen, wobei
       die Adresse von help_c der von var_l gleicht */
9 int longAlign = &long_str.un.help_c - &long_str.var_c;
```

Beim Datentyp `char` ist zwar die Größe mit einem Byte und das Alignment mit Eins vorgegeben, jedoch nicht, ob er `signed` oder `unsigned` interpretiert wird. Zur Überprüfung wird ein negativer Wert in eine `char`-Variable geschrieben und dann getestet, ob die Variable kleiner Null ist.

```
1 char var_c = (char)(-3);
2 if (var_c < 0)
3 ...
```

5.2.2.4 Enum

Referenz 3.1.9
Wie in Abschnitt 3.1.9 beschrieben und die Ergebnisse in Kapitel 6 zeigen, unterscheidet sich das Verhalten von Enumerations bei verschiedenen Compilern sehr stark voneinander. Deshalb wird zur Überprüfung deren Größe ein eigener Test durchgeführt, in dem von `char` bis `long long` alle Möglichkeiten durchgespielt werden. Dabei gibt der Kommentar im Codeausschnitt an, wie groß der jeweilige Datentyp mindestens sein muss. Auch hier wird während des Kompilierens erst überprüft, welche der aufgelisteten Möglichkeiten überhaupt akzeptiert werden. In diesem Beispiel wird das Makro `RB_CEVAL_ENUM_64BIT` nur dann gesetzt, wenn der Kompiliervorgang mit `enum listA` und `enum listB` auch erfolgreich war. Auf das Prefix `rbCEval` wird aus Platzgründen verzichtet.

```
1  enum list1 {a1,          b1,          c1};/* 8 Bit          */
2  enum list2 {a2 = -1,     b2,          c2};/* 8 Bit   signed */
3  enum list3 {a3 = 128,    b3,          c3};/* 8 Bit unsigned */
4  enum list4 {a4 = -1,     b4 = 128,    c4};/*16 Bit   signed */
5  enum list5 {a5 = 256,    b5,          c5};/*16 Bit          */
6  enum list6 {a6 = 32768,  b6,          c6};/*16 Bit unsigned */
7  enum list7 {a7 = -1,     b7=32768,    c7};/*32 Bit   signed */
8  enum list8 {a8 = 65536,  b8,          c8};/*32 Bit          */
9  enum list9 {a9 = 2147483648, b9,      c9};/*32 Bit unsigned */
10 #ifdef RB_CEVAL_ENUM_64BIT
11 enum listA {aA= -1, bA=2147483648, cA};/*64 Bit   signed */
12 enum listB {aB = 4294967296, bB,    cB};/*64 Bit          */
13 #endif
```

Anhand von `list1` zeigt der nächste Ausschnitt, wie die Größe des jeweiligen Enums festgestellt wird, ob der Datentyp `signed` oder `unsigned` ist und wie viele Bytes ein einzelnes Element belegt. Bei einem Element handelt es sich um eine Konstante, die auch unabhängig vom Enum verwendet werden kann. Deren Größe kann von der Größe des Enums abweichen .

```
1 enum list1 var1_e = a1;
2 /* Größe von list1 in Byte */
3 char list1Size_c = sizeof(var1_e);
4 /* Ist der Datentyp von list1 signed(=1) oder unsigned(=0)? */
5 char list1Sign_c = 0;
6 /* welche größe hat ein Element */
7 char list1ElementSize_c = 0;
8 var1_e = ~var1_e;  /* setzt alle Bits auf 1 */
```

```
9  if(var1_e < 0)
10   list1Sign_c = 1;
11 list1ElementSize_c = sizeof(a1);
```

Der Anwender bekommt in der Auswertung dann eine Liste mit den überprüften Enumerations, der Angabe, wie viele Bytes diese belegen, ob der Datentyp `signed` oder `unsigned` ist und wie viele Bytes ein einzelnes Element belegt.

5.2.2.5 Modulo und Division von Integerzahlen

Referenz 3.1.5, 3.1.13
Laut Standard C90 hängen die Integeroperationen Division und Modulo über die in Abschnitt 3.1.5 angegebene Formel zusammen. Es ist möglich, dass ein Compiler diesen Zusammenhang nicht korrekt implementiert. Trotzdem interessiert in diesem Fall das Verhalten der beiden Operationen. Deshalb findet zuerst eine Überprüfung der Formel für alle vier möglichen Kombinationen statt mit $(a, b) \in \{(9, 5), (9, -5), (-9, 5), (-9, -5)\}$. Ist diese Formel nicht für alle Kombinationen erfüllt, wird eine Warnung ausgegeben.

```
1 if (((a/b)*b + a%b) != a)
2         /* Gleichung nicht erfüllt */
```

Ergebnis der Division

Das Ergebnis der Division mit positiven Operanden ist definiert als die nächste ganze Zahl, die kleiner oder gleich dem Quotienten ist. Wie in Abschnitt 3.1.5 gezeigt, gibt es bei negativen Zahlen drei Kombinationen mit jeweils zwei Ergebnissen, wobei `9/-5` und `-9/5` das gleiche Ergebnis haben müssen, entweder `-1` oder `-2`. Diese drei Kombinationen werden folgendermaßen analysiert:

```
1 int result_i = 0;
2 if(9/-5 == -2)
3     result_i += 1;
4 if(-9/5 == -2)
5     result_i += 2;
6 if(-9/-5 == 2)
7     result_i += 4;
```

Es gibt acht mögliche Kombinationen, wobei nur vier dem Standard entsprechen. Bei den anderen vier wird eine Warnung ausgegeben.

Modulo mit negativen Zahlen

Da angenommen wird, dass die Implementierung von Modulo vom Standard C90 abweichen könnte, müssten alle Möglichkeiten überprüft werden. Durch die beiden vorherigen Analysen, ist bereits bekannt, ob die Implementierung dem Standard entspricht und welche Ergebnisse zu erwarten sind. Es findet deshalb nur eine Überprüfung des Vorzeichens statt, wobei alle acht theoretischen Möglichkeiten aus Abschnitt 3.1.5 in Betracht gezogen werden.

```
1 int result_i = 0;
2 if ( 9 % (-5) < 0)
3   result_i += 1;
4 if ((-9) % (-5) < 0)
5   result_i += 2;
6 if ((-9) % 5 < 0)
7   result_i += 4;
```

Die Variable `result_i` hat danach einen Wert von 0 bis 7, so dass alle acht Möglichkeiten abgedeckt sind. Das Gleiche wird in ähnlicher Form für den Präprozessor durchgeführt.

Division und Modulo mit Null

Viele Compiler fangen eine Division durch eine Nullkonstante direkt ab (`x = 3 / 0`). Das heißt, es werden für beide Operationen zwei Tests durchgeführt. Einerseits, ob der Compiler eine ungültige Operation mit einer Nullkonstante erkennt und einen Fehler ausgibt, und andererseits, wie sich das Programm zur Laufzeit verhält. Die Untersuchung während des Programmablaufs kann nicht über eine Konstante erfolgen. Hier wird eine mit `volatile` deklarierte Variable auf Null gesetzt und durch diese dividiert. Die Angabe von `volatile` zeigt dem Compiler, dass sich die Variable ohne Einfluss der Software ändern kann und deshalb nicht wegoptimiert werden darf. Wegen der Gefahr eines Programmsabsturzes müssen diese Tests getrennt von den anderen stattfinden, da ein Absturz auch Auswirkungen auf deren Ergebnisse haben könnte. Dies ist einer der Gründe, warum mehrere Programmläufe nötig sind.

5.2.2.6 Bitverschiebung

Referenz 3.1.5
Um herauszufinden, ob eine Bitverschiebung nach rechts bei `signed` Werten arithmetisch oder logisch stattfindet, wird wie in 3.1.5 beschrieben, eine `signed int` Variable mit -1

initialisiert und um ein Bit nach rechts verschoben. Ist ihr Wert danach immer noch -1, handelt es sich um eine arithmetische Verschiebung.

```
1 signed int var_si = -1;
2 var_si >>= 1;
3
4 if (var_si == -1){
5   /* arithmetisch mit dem MSB aufgefüllt */
6 }else{
7   /* logisch, mit 0 aufgefüllt */
8 }
```

5.2.2.7 Datentypkonvertierung

Referenz 3.1.5
Die Untersuchung des Verhaltens bei der Konvertierung zu kleineren `signed`-Datentypen geht auf die folgenden Möglichkeiten ein:

1. `unsigned char` → `signed char`
   Ein zu großer positiver Wert vom gleichen Typ soll in einem `signed char` gespeichert werden.

2. `signed long` → `signed char`
   Ein zu großer negativer Wert eines größeren Typen soll in einem `signed char` gespeichert werden.

3. `unsigned long` → `signed char`
   Ein zu großer positiver Wert eines größeren Typen soll in einem `signed char` gespeichert werden.

Zur Untersuchung wird ein bestimmtes Bitmuster in drei Variablen des jeweils ersten Typen geschrieben, diese werden dann zu `signed char` konvertiert und das Bitmuster wird analysiert. Dabei wird das Zweierkomplement vorausgesetzt.
Auch wenn der Standard C90 es offen lässt, sollte sich das Bitmuster nicht ändern, da dies zusätzliche interne Berechnungen zur Folge hätte. Deshalb wird nur überprüft, ob das Bitmuster gleich bleibt. Das Codebeispiel zeigt die Analyse der ersten Möglichkeit.

```
1 unsigned char var_uc = 129; /* Bitmuster: 10000001 */
2 signed char var_sc = (signed char) var_uc;
3 if(var_c == -127){
4   ... /* Bitmuster unverändert */
```

```
5 }else{
6    ... /* Warnung, Bitmuster hat sich verändert */
7 }
```

5.2.2.8 Strukturen

Referenz 3.1.9
Zusätzlich zum Alignment der einzelnen Datentypen, dessen Überprüfung in Abschnitt 5.2.2.3 beschrieben wird, findet eine Untersuchung der Größe von Strukturen statt. Es geht darum, ob sich die Größe bei einer unterschiedlichen Reihenfolge der Elemente ändert.

```
1 struct {
2   char var_c;
3   int  var_i;
4 } ci_str;
5
6 struct {
7   int  var_i;
8   char var_c;
9   } ic_str;
```

Hat `int` in diesem Beispiel ein gerades Alignment, wird in der ersten Struktur `ci_str` ein Füllbyte verwendet und die gesamte Struktur benötigt 4 Byte. Untersucht wird, ob auch bei der zweiten Struktur `ic_str` nach dem `char` noch ein Füllbyte hinzugefügt wird und sich damit die Größe nicht geändert hat. Da sich der `sizeof`-Operator auf den Abstand zweier Elemente eines Arrays bezieht, kann er in diesem Fall nicht verwendet werden. Damit `var_i` auch in einem Array immer an einer geraden Adresse liegt, muss die Struktur `ic_str` ebenfalls die Größe 4 Byte haben.
Laut Standard müssen die einzelnen Elemente einer Struktur, abgesehen von Füllbytes, in aufeinanderfolgenden Speicherbereichen liegen ([ISO90]:6.5.2.1). Untersucht wird deshalb eine verschachtelte Struktur, bei der nach der inneren Struktur ein `char` folgt.

```
1 stuct{
2   struct{
3     int  var_i;
4     char var1_c;
5   } intern_str;
6   char var2_c;
7 } test_str;
```

Ist die Differenz der Adressen der beiden `char`-Elemente größer als 1, wird ein Füllbyte verwendet. Dies wird in der folgenden Abbildung 5.1 veranschaulicht.

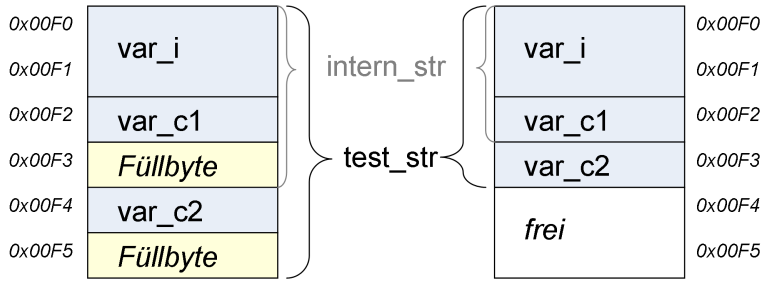


Abbildung 5.1: Füllbyte am Ende von Strukturen

5.2.2.9 Endianess

Referenz: 3.1.9
Mit Hilfe von Unions und Arrays kann die Anordnung der Bytes von großen Datentypen überprüft werden. Da der Datentyp `char` immer ein Alignment von 1 hat, gibt es keine Füllbytes. Somit bildet ein `char`-Array den internen Speicher Byte für Byte ab. Wird nun ein `char`-Array mit einer Variable vom Datentyp `long` kombiniert, lässt sich die interne Verteilung der Bytes in einem `long` herausfinden, wie das folgende Beispiel zeigt:

```
1  union{
2    char array_c[sizeof(long)];
3    long var_l;
4  } endian_u;
5  /* Array zum speichern der Byteanordnung */
6  char order_c[sizeof(long)];
7  int i,j;
8  /* setze das erste Bit */
9  endian_u.var_l = 1;
10
11 for(i = 0; i < sizeof(long); i++){
12   order_c[i] = 0;
13   /* suche das Byte in dem das Bit gesetzt wurde */
14   for(j = 0; j < sizeof(long); j++){
15     if(endian_u.array_c[j] == 1)
16       order_c[i]= j + 1;
17   }
18   /* verschiebe das Bit um 1 Byte */
19   endian_u.var_l <<= 8;
20 }
```

Es wird ein `union endian_u` erzeugt, das ein `char[sizeof(long)] array_c` und ein `long var_l` enthält. Laut Definition eines `union`(C90) überlappen sich die Speicherbe-

reiche von `array_c` und `var_l` genau. Wird nun ein Bit von einem Byte der `long`-Variable `var_l` zum nächsten geschoben (Zeile 19) und verglichen, in welchem der Elemente des `char`-Arrays es sich dann befindet (Zeile 14-17), erhält man die interne Anordnung der Bytes.
Am Ende enthält das Array `order_c` die Kombination `{1,2,3,4}` für Little-Endian und `{4,3,2,1}` für Big-Endian.

5.2.2.10 Bitfelder

Referenz 3.1.9, 3.2.2
Bei Bitfeldern werden die folgenden Punkte untersucht:

1. Abhängigkeit der Größe von den genutzten Bits
2. Anordnung der einzelnen Bits
3. Alignment (Verwendung von Füllbits)
4. Signedness von einfachen `int`-Bitfeldern und `signed int`-Bitfeldern
5. Datentypen von Bitfeldern

Die Erklärungen beziehen sich auf ein System, in dem `int` 16 Bit groß ist. Sie sind aber auch für andere Systeme anwendbar. Die Analysen finden mit Hilfe von Strukturen und Unions statt.

Größe

Ob die Größe eines Bitfeldes von den genutzten Bits abhängt, wird geprüft, indem ein Bitfeld mit nur einem Bit erzeugt wird. Ist dessen mit `sizeof`(Bitfeld) ermittelte Größe kleiner als die Größe eines `int`, optimiert der Compiler die Größe automatisch.

Anordung

Mit einem weiteren Schritt kann auch die Anordnung der Bits geprüft werden. Hierzu wird ein Union genutzt. Mit dessen Hilfe lässt sich der Wert des Bitfeldes über eine `int`-Variable (oder bei Optimierung über eine `char`-Variable) auslesen. Ist dieser Wert 1, wurde das LSB gesetzt, ist er 2^{15} (2^7), liegt das erste Bit auf dem MSB (siehe auch Abbildung 3.5).

Eine mögliche Verschiebung der Bits bei einer Änderung des Datentypen, wie in Abbildung 3.6 beschrieben, kann mit dem folgenden Code überprüft werden.

```
1  /* nur ein Bit wird genutzt */
2  union {
3      struct{
4          unsigned int b0:1;
5      }bits;
6      unsigned int var_i;
7  } bitFeld1Bit_un;
8  /* 9 Bit werden genutzt */
9  union {
10     struct{
11         unsigned int b0:1;
12         unsigned int b0:8;
13     }bits;
14     unsigned int var_i;
15 } bitFeld9Bit_un;
16 /* setze alles zu 0 */
17 bitFeld1Bit_un.var_i = 0;
18 bitFeld9Bit_un.var_i = 0;
19 /* setze das erste Bit */
20 bitFeld1Bit_un.bits.b0 = 1;
21 bitFeld9Bit_un.bits.b0 = 1;
22 /* vergleiche die int-Werte */
23 if(bitFeld1Bit_un.var_i != bitFeld9Bit_un.var_i)
24 ...
```

Zuerst werden alle Werte in beiden Unions auf 0 gesetzt (Zeile 17/18). Dann wird in beiden Bitfeldern jeweils nur das niederwertigste Bit auf 1 gesetzt (Zeile 20/21), und die beiden Integerwerte verglichen, Zeile 23. Stimmen sie nicht überein, hat sich die Anordnung der Bits geändert und es wird eine Warnung ausgegeben.

Alignment

Was es mit dem Alignment in Bitfeldern auf sich hat, wird in Abbildung 3.4 beschrieben. Um herauszufinden, ob ein Bitfeld auf eine Einheit begrenzt ist oder zwei Einheiten überlappen kann, wird die Größe der folgenden Struktur bestimmt:

```
1 struct{
2   int bits1:6;
3   int bits2:16;
4   int bits3:10;
5 }bitFeld_str;
```

Insgesamt werden nur 32 Bits verwendet. Ist `sizeof(bitFeld_str)` gleich 4, werden keine Füllbits verwendet (Abbildung 3.4, rechts). Bei `sizeof(bitFeld_str)` gleich 6 werden Füllbits genutzt, wie auf der linken Seite der Abbildung 3.4 dargestellt.

Signedness

Für die Analyse, ob ein einfaches `int`-Bitfeld als signed oder unsigned interpretiert wird, werden alle Bits eines `int`-Bitfeldes auf 1 gesetzt und überprüft, ob der Wert positiv oder negativ ist.

Mögliche Datentypen

In den „Common Extensions“ der „Portability Issues“ (vgl. Anhang A) wird erwähnt, dass Bitfelder auch mit anderen Datentypen als `int` deklariert werden können. Deshalb wird weiterhin überprüft, welche Datentypen vom Compiler für Bitfelder zugelassen werden. Dies geschieht während der Compilerfehleranalyse, da beispielsweise ein `long`-Bitfeld zu einem Compilerfehler führen kann.

5.2.2.11 Gleitkommazahlen

Referenz 3.1.6
Bei den Gleitkommazahlen muss untersucht werden, wie sie intern dargestellt werden, wie präzise gerechnet wird und wie das Rundungsverhalten ist. Für die interne Darstellung ist die Erfüllung der Formel aus Abschnitt 3.1.6 ausschlaggebend.

$$x = s * b^e * \sum_{k=1}^{p} f_k * b^{-k}$$

mit:

- x = normalisierte Gleitkommazahl ($f_1 > 0, x \neq 0$)
- s = sign (± 1)
- b = Basis
- p = Präzision, Anzahl der Basis b Stellen im signifikanten Teil
- e = Exponent (Integer zwischen $e_{min} \leq e \leq e_{max}$)
- f_k = nicht negatives Integer kleiner als b

Die folgenden Berechnungen basieren zum Teil auf Algorithmen, die dem Programm „paranoia.c“ entnommen sind.[5]

[5] C Programm für umfassende Tests der Gleitkommaarithmetik. [This program is credited to W. M. Kahan, B. A. Wichmann, David M. Gay, Thos Sumner, and Richard Karpinski.] `http://www.netlib.org/paranoia`

Interne Darstellung

s belegt in jedem Fall 1 Bit, da es nur das Vorzeichen angibt.
Die Basis b wird anhand der folgenden Schritte berechnet:

1. Bestimme den `float`-Wert W so, dass $W + 1$ nicht ohne Genauigkeitsverlust abgespeichert werden kann.
2. Erzeuge einen weiteren `float`-Wert $Z = W + Y$, wobei Y soweit erhöht wird, bis die Differenz $Z - W \neq 0$ ergibt.
3. Die Basis ist dann gleich dieser Differenz: $b = Z - W$

Zur besseren Lesbarkeit des Codes wird `base` statt `b` verwendet (sowie im nächsten Ausschnitt `precision` statt `p`).

```
1  volatile float base, W, Y, Z;
2  W = 1.0f;
3
4  /* Schritt 1 */
5  do{
6       W = W + W;
7       Y = W + 1.0f;
8       Z = Y - W;
9       Y = Z - 1.0f;
10 } while (-1.0f + abs(Y) < 0.0f);
11 /*... jetzt ist W groß genug damit |((W+1)–W)–1| >= 1 */
12 Y = 1.0f;
13 /* Schritt 2./3. Finde die Basis b */
14 do{
15      Z = W + Y;
16      Y = Y + Y;
17      base = Z - W;
18 } while ( base == 0.0f);
```

Die Präzision p gibt die Anzahl der möglichen Nachkommastellen zur Basis b an. Sie wird mit dem folgenden Algorithmus berechnet, bei dem die Basis solange mit sich selbst multipliziert wird, bis

$$(base^{precision} + 1.0) - base^{presision} \neq 1.0$$

```
1  int precision;
2  volatile float W ,Y;
3
4  precision = 0;
5  if (base != 1)
6  {
7      W = 1.0f;
8      do{
9          precision++;
10         W = W * base;    /* = base^precision */
11         Y = W + 1.0f;
12     } while ((Y - W) == 1.0f);
13 }
14 /*... jetzt ist W == base^precision gerade zu groß um (W+1)-W
       == 1 zu erfüllen */
```

Die Berechnung von p wird für jede der drei Gleitkommazahlen `float`, `double` und `long double` nach dem gleichen Schema durchgeführt. Bei dem Ergebnis ist eine Besonderheit zu beachten, die in den meisten Fällen auftreten wird:
Ist die Basis $b = 2$ gilt laut Definition immer $f_1 = 1$. Deshalb wird in diesem Fall, um Platz zu sparen, das erste Bit nicht gespeichert. Ist die Präzision p also beispielsweise 24, ist es ausreichend, nur die 23 niederwertigen Bits der Mantisse zu speichern.

Die Größe von dem Exponenten e wird im Nachhinein in der Auswertung nach der folgenden Formel berechnet:

$$e = \begin{cases} \text{Bits} - (p-1) - s & \text{für } b = 2 \\ \text{Bits} - p - s & \text{für } b > 2 \end{cases}$$

Genauigkeit

Zusätzlich zur Mantisse, die die Nachkommastellen zur Basis b angibt, wird noch die Genauigkeit der drei Datentypen in Nachkommastellen zur Basis 10 berechnet. Dazu wird zu 1.0 ein Epsilon addiert und überprüft, ob die Summe sich von 1.0 unterscheidet. Das Epsilon wird nach und nach verkleinert (10^{-1}, 10^{-2}, ...). Dies ist keine genaue Berechnung und soll dem Anwender nur einen Richtwert geben. Ausgegeben wird deshalb nur der negative Exponent.

Rundungsverhalten

Für das Rundungsverhalten von Gleitkommazahlen bei Werten, die nicht genau darstellbar sind, gibt der Standard die folgenden vier Möglichkeiten vor:

- zu Null
- zur nächsten Nachkommastelle
- zu plus unendlich
- zu minus unendlich

Es dürfen zwar weitere Möglichkeiten definiert werden, es ist jedoch nicht sinnvoll, nach diesen zu suchen, da die wichtigsten hierdurch abgedeckt werden.
Das Rundungsverhalten wird für zwei Fälle untersucht. Erstens, wenn eine Dezimalzahl im binären System nicht eindeutig darstellbar ist, und zweitens, wenn ein `long`-Wert in einen `float`-Wert konvertiert wird und dieser sich nicht genau darstellen lässt.
Als Dezimalzahl wird 0.1 genommen, da sich bei ihrer Darstellung als Binärzahl eine Periode ergibt :

$$0.1_{10} = 0.0001100\overline{1100}_2$$

Die nächsten Dezimalwerte, die sich durch eine 32-Bit `float`-Zahl darstellen lassen, sind in etwa

$0.099999994039535522460937 5_{10}$
$0.100000001490116119384765625_{10}$

Für die Überprüfung des Rundungsverhaltens werden Werte nahe der möglichen Werte in `float`-Variablen geschrieben und dann mit den genauen Werten verglichen (für positive und negative Zahlen). Anhand dieses Vergleichs lässt sich ableiten, wie gerundet wird. Das folgende Beispiel zeigt den Vergleich mit vereinfachten Zahlen. Die nächsten zu 0.1 darstellbaren Zahlen seien 0.099 und 0.101. Links stehen die angenäherten Werte und rechts die genauen:

```
1 if ((float)((float) 0.0995 - (float) 0.099) == 0 &&
2     (float)((float) -0.0995 - (float) -0.099) == 0 &&
3     (float)((float) 0.1005 - (float) 0.099) == 0 &&
4     (float)((float) -0.1005 - (float) -0.099) == 0 )
```

Wenn diese vier Bedingungen wahr sind, wird immer zu Null gerundet, also immer zu ± 0.099.

5.2.2.12 Zeichensatz

Referenz 3.1.4, 3.1.13
Wie bereits erwähnt, wird meist ASCII als Zeichensatz verwendet. Dabei wird der ASCII-

Zeichensatz nochmals in den ursprünglichen 7-Bit und den erweiterten 8-Bit Zeichensatz unterteilt. Für die Überprüfung des verwendetetn Zeichensatzes werden drei Unterscheidungen gemacht. Zuerst werden alle Zeichen überprüft, die der Standard vorsieht (siehe Auflistung in Abschnitt 3.1.4). Diese werden vollständig durch den 7-Bit ASCII-Zeichensatz abgedeckt.
Übrig vom 7-Bit Zeichensatz bleiben dann noch diverse, nicht darstellbare Steuerzeichen und die folgenden drei Zeichen:

```
$ @ ‘ [6]
```

Die Zeichen `$` und `@` werden bei manchen Compilern verwendet, um compilerspezifische Befehle zu kennzeichnen (@tiny, @near, @far beim Cosmic Compiler [Cos05]). `$` kann Teil von speziellen Linker-Adressen sein.
Übrig bleiben die restlichen Zeichen des erweiterten 8-Bit ASCII-Zeichensatzes (ASCII-Code 128-255). Diese gehören zwar nicht zum Standard, werden aber der Vollständigkeit halber ebenfalls überpüft. Zusätzlich wird der Standardzeichensatz für den Präprozessor getestet. Verwendet dieser ebenfalls ASCII, liefern `if`-Konstrukte, wie in Abschnitt 3.1.13 gezeigt, das gleiche Ergebnis.
Die Überprüfung findet durch den Vergleich der Zeichenkonstanten mit ihrem dezimalen ASCII-Wert statt:

```
1 int counter_i = 0;
2 if ('a' != 97) counter_i++;
3 if ('b' != 98) counter_i++;
4 ...
```

Entsprechend für den Präprozessor:

```
1 int counter_i++;
2 #if 'a' != 97
3   counter_i++;
4 #endif
5 #if 'b' != 98
6 ...
```

Am Ende jeder Überprüfung gibt die Variable `counter_i` die Abweichungen vom ASCII-Zeichensatz an. Ist ihr Wert Null, werden alle überprüften Zeichen in ASCII dargestellt. Gibt es Abweichungen im Standardzeichensatz, bekommt der Anwender bei der Auswertung einen Hinweis.

[6] Hier ist nicht das Hochkomma (') mit ASCII-Code 39 gemeint sondern (‘) mit ASCII-Code 96.

5.2.2.13 Bezeichner

Referenz: 3.1.3
Die Idee ist die dynamische Generierung von Bezeichnern mit gleicher Länge, bei denen sich nur die letzte Stelle des Namens unterscheidet. Diese werden dann in ein Stück Code eingebunden und kompiliert. Werden die Namen akzeptiert, folgt der nächste Durchlauf mit längeren Bezeichnern. Kommt es zu einem Compilerfehler, werden für den nächsten Durchlauf kürzere Namen generiert. Um einen möglichst großen Namensraum mit wenigen Durchläufen abzudecken, wird das Prinzip der binären Suche angewendet, wie in Abbildung 5.2 dargestellt. Die obere Grenze wird dabei auf 1024 (2^{10}) festgesetzt. Das heißt, Bezeichner mit einer Länge von 0 - 1023 werden untersucht. Dazu werden 10 Durchläufe benötigt.

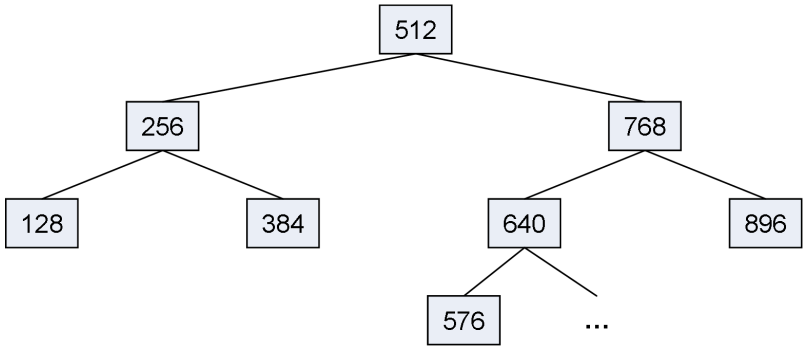


Abbildung 5.2: Binäre Suche der Bezeichnerlänge

Wie schon in Abschnitt 3.1.3 erwähnt, sind dabei für externe Bezeichner mehrere Szenarien zu beachten. Diese werden in vier Schritten umgesetzt.

1. Zwei globale Variablen definiert in einer Datei. (10 Durchläufe)
2. Zwei globale Variablen mit der maximalen Länge aus dem ersten Schritt definiert in einer Datei und Zugriff auf diese in einer zweiten Datei zur Überprüfung der gefundenen Länge. (1 Durchlauf)
3. Eine globale Variable in einer Datei, um zu überprüfen, ob der Compiler den Namen auf den signifikanten Teil reduziert. (10 Durchläufe)
4. Zwei globale Variablen in verschiedenen Dateien und deren Aufruf in einer dritten Datei, um den Linker zu überprüfen. (10 Durchläufe)
   Die maximal erlaubte Länge aus diesem Schritt wird später auch im Programmlauf verwendet, um zu überprüfen, ob die Variablen tatsächlich unterschieden werden. Dazu wird deren Inhalt während des Programmlaufs untersucht. Ist der Inhalt gleich,

wurden die Variablen vom Linker nicht unterschieden. Das heißt, entweder wurde die zweite Datei nicht gelinkt, weil alle Abhängigkeiten durch die erste Variable erfüllt sind, oder der Wert der ersten Variable wurde von der zweiten, später gelinkten Variable überschrieben.

Der erste Schritt wird auch für Makronamen durchgeführt. Da Makronamen durch den Präprozessor verarbeitet werden, sind keine weiteren Schritte notwendig.

5.2.2.14 Inhalt von Makros

Referenz: 3.2.3
Der Inhalt eines Makros ließe sich anhand der Präprozessorausgabe ermitteln. Dazu wird die Ausgabe an der Stelle durchsucht, an der ein Makro im Quellcode stand und prüft dort wodurch das Makro ersetzt wurde. Dieser Prozess lässt sich jedoch nicht sehr gut automatisieren. Um den Code nach dem Präprozessor zu erhalten, müssen üblicherweise auch andere Compileroptionen gesetzt werden, die in einem Projekt ansonsten nicht verwendet werden.
Der Präprozessor selbst bietet mit dem „stringize"-Operator `#` eine andere Möglichkeit, den Inhalt eines Makros darzustellen. Der Standard C90 beschreibt im Abschitt 6.8.3.5 ein Beispiel, wie dieser Operator angewendet werden kann. Dazu wird ein funktionsähnliches Makro erstellt, das den übergebenen Parameter während des „Preprocessing" in einen String umwandelt, also noch bevor der eigentliche Quellcode übersetzt wird.

```
#define RB_CEVAL_STRINGIZE(a) #a
printf(RB_CEVAL_STRINGIZE(Hello world!));
```

Dieser Code sieht nach dem „Preprocessing" folgendermaßen aus:

```
printf("Hello world!");
```

Mit dem oben definierten `RB_CEVAL_STRINGIZE(a)` wird allerdings nur `a` und nicht der Inhalt von `a` in einen String umgewandelt. Dazu ist ein weiterer Schritt notwendig. Dieser nutzt die Reihenfolge aus, mit der der Präprozessor die einzelnen Makros auflöst. Es wird deshalb ein zweites Makro definiert:

```
#define RB_CEVAL_MACRO_CONTENT(b) RB_CEVAL_STRINGIZE(b)
```

Die schrittweise Auflösung während des „Preprocessing" wird am Beispiel des Makros `NULL` dargestellt:

```
1 rbCEval_outputString(RB_CEVAL_MACRO_CONTENT(NULL));
2 rbCEval_outputString(RB_CEVAL_STRINGIZE(NULL));
3 rbCEval_outputString(RB_CEVAL_STRINGIZE((void *)0);
4 rbCEval_outputString("(void *)0");
```

Zuerst wird in Zeile 2 `RB_CEVAL_MACRO_CONTENT` zu `RB_CEVAL_STRINGIZE` aufgelöst, wobei der Parameter weitergegeben wird. Im nächsten Schritt wird das Makro `NULL` mit dessen Definition ersetzt, in diesem Fall `(void *)0`. Zum Schluss wird das andere Makro aufgelöst. In Zeile 4 steht der Code nach dem Preprocessing, der dann kompiliert und ausgeführt wird.
Damit gibt es eine Möglichkeit, den Inhalt von Makros zu erhalten, ohne dass der Quellcode nach dem Präprozessor analysiert werden muss.

5.3 Automatisierung

Nachdem die Analysen festgelegt sind, müssen diese nun so weit wie möglich automatisiert durchgeführt werden. Als Grundlage dafür dient das in Abschnitt 4.3 erstellte Datenmodell. Die Verarbeitung und Erzeugung von Daten wird dort in vier Prozesse unterteilt (siehe auch Abbildung 4.2).

1. Die dynamische Codegenerierung
2. Der Build-Prozess
3. Der Programmlauf
4. Die Auswertung

Die dynamische Codegenerierung und die Auswertung können automatisiert werden. Der Build-Prozess muss vom Anwender bereitgestellt werden, kann dann aber vom Programmsystem automatisiert verwendet werden. Der Programmlauf auf dem Mikroprozessor muss manuell durchgeführt werden. Der Aufbau des Programmsystem muss also so entwickelt werden, dass der Build-Prozess und der Programmlauf optimal eingebunden werden.

Bevor jedoch auf die einzelnen Prozesse näher eingegangen wird, wird zum besseren Verständnis der folgenden Erklärungen das Resultat grafisch dargestellt. Das Diagramm in der Abbildung 5.3 zeigt den Ablauf des Programmsystems *CEval* aus Sicht des Anwenders. Es wird dargestellt, welche Aufgaben vom Anwender durchzuführen sind und in welcher Reihenfolge die Perlskripte gestartet werden müssen, um die automatisierten Prozesse zu starten.

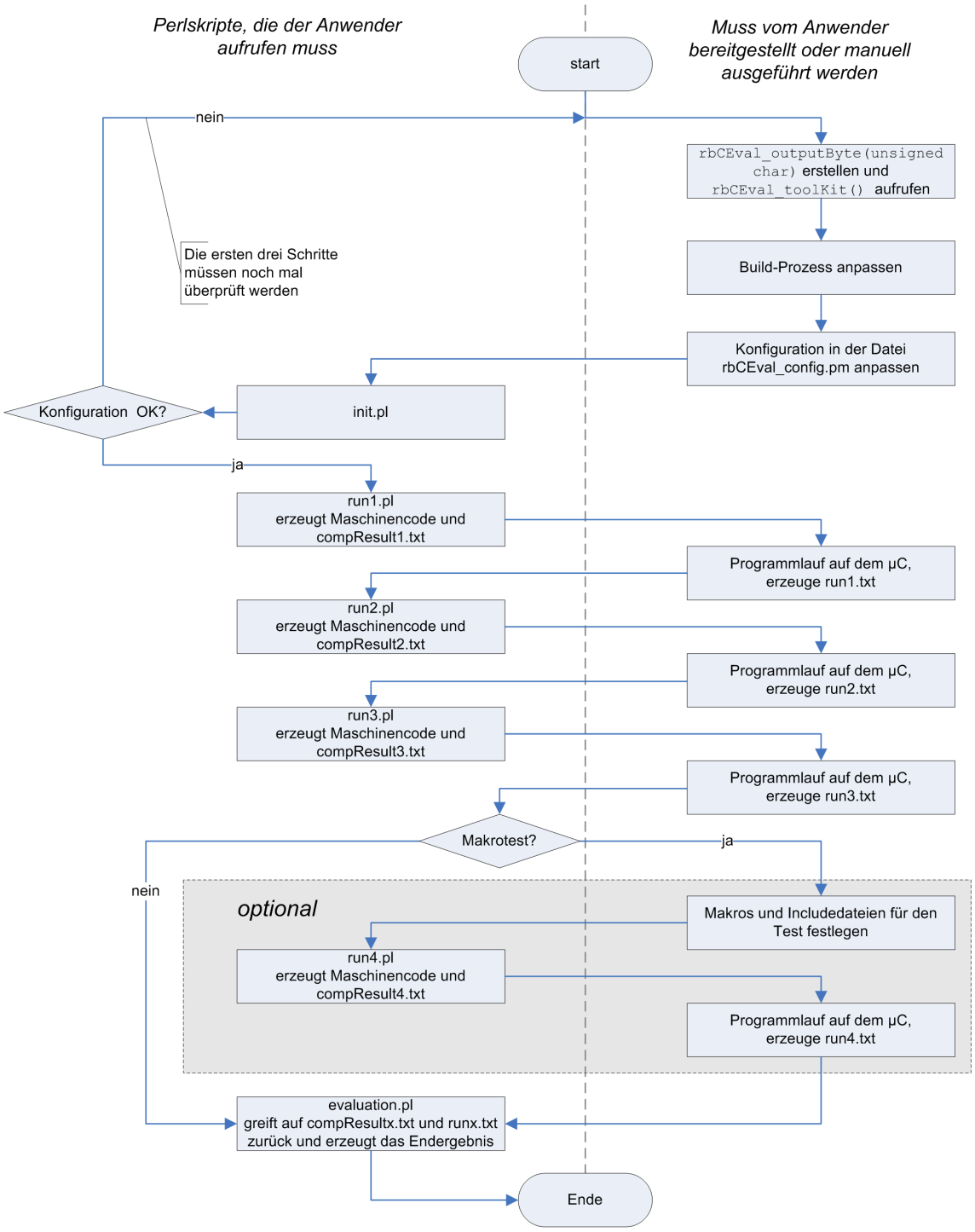

Perlskripte, die der Anwender aufrufen muss
Muss vom Anwender bereitgestellt oder manuell ausgeführt werden
start
nein
rbCEval_outputByte(unsigned char) erstellen und rbCEval_toolKit() aufrufen
Die ersten drei Schritte müssen noch mal überprüft werden
Build-Prozess anpassen
Konfiguration in der Datei rbCEval_config.pm anpassen
Konfiguration OK?
init.pl
ja
run1.pl erzeugt Maschinencode und compResult1.txt
Programmlauf auf dem µC, erzeuge run1.txt
run2.pl erzeugt Maschinencode und compResult2.txt
Programmlauf auf dem µC, erzeuge run2.txt
run3.pl erzeugt Maschinencode und compResult3.txt
Programmlauf auf dem µC, erzeuge run3.txt
Makrotest?
ja
nein
optional
Makros und Includedateien für den Test festlegen
run4.pl erzeugt Maschinencode und compResult4.txt
Programmlauf auf dem µC, erzeuge run4.txt
evaluation.pl greift auf compResultx.txt und runx.txt zurück und erzeugt das Endergebnis
Ende

Abbildung 5.3: Gesamtablauf

5.3.1 Anpassungen am Build-Prozess

Am Beispiel des Tools Make werden hier die vom Anwender vorzunehmenden Anpassungen erklärt.
In der Make-Konfigurationsdatei sind zusätzlich zu den Parametern und Optionen für den Compiler und den Linker die Abhängigkeiten der verschiedenen Quellen hinterlegt. Das bedeutet, dass es eine Liste mit C-Dateien gibt, die zu dem Projekt gehören und kompiliert werden müssen. Neben den Dateien werden normalerweise auch die Pfade, in denen die Dateien zu finden sind, übergeben (Include-Pfade). Diese Aufstellung gilt nicht nur für Make, sondern in dieser oder ähnlicher Form auch für andere Build-Prozesse (Perlskript, Batchdatei, ...).
Da die Analyse mehrere Durchläufe benötigt, diese aber alle mit demselben Build-Prozess durchgeführt werden sollen, muss ein Weg gefunden werden, dem Build-Prozess mitzuteilen, welche Dateien zum aktuellen Durchlauf gehören und kompiliert werden sollen.
Statt nun bei jedem Durchlauf neue Dateienamen zu übergeben, werden mehrere Dateienamen festgelegt und deren Inhalt vor jedem Durchlauf angepasst.
Während der dynamischen Codegenerierung werden für jeden Durchlauf insgesamt sechs Dateien mit dem Namen `rbCEval_x.c` erstellt, wobei das `x` für eine Zahl von 0 bis 5 steht. Diese Dateien muss der Anwender in seinen Build-Prozess einbinden.
Neben diesen sechs Dateien muss es mindestens noch eine weitere Datei geben, die die Startfunktion (normalerweise `main()`) und die Funktion `rbCEval_outputByte()` zur Ausgabe von den Ergebnissen enthält (siehe Abschnitt 4.4.4).
Bevor jedoch diese sechs Dateien eingebunden werden, muss der Anwender sicherstellen, dass der von ihm bereitgestellte Quellcode ohne Fehler kompiliert, andernfalls kann der nächste Schritt nicht funktionieren.

Der nächste Schritt dient der Erkennung von Compilerfehlern ohne die Analyse der Fehlermeldungen, wie in Abschnitt 4.4.2 beschrieben. Zum Ende eines erfolgreichen Build-Prozesses muss dazu die Datei `rbCEval_compile.txt` in ein temporäres Verzeichnis geschrieben werden. Ist dagegen der Build-Prozess nicht erfolgreich, darf diese Datei nicht erzeugt werden. Wenn der Quellcode so generiert wurde, dass voraussichtlich nur an einer Stelle ein Fehler auftreten kann, können gezielt Eigenschaften untersucht werden, die je nach Implementierung zu Compilerfehlern führen.
Der Build-Prozess wird in der Compilerfehleranalyse wie eine Funktion genutzt, die anhand des Quellcodes in den Dateien `rbCEval_x.c` „wahr“ oder „falsch“ zurückgibt. Das „wahr“ ist in diesem Fall das Vorhandensein der Datei `rbCEval_compile.txt` und „falsch“ deren Abwesenheit. Der gesamte Ablauf des Build-Prozesses ist in der Abbildung 5.4 vereinfacht dargestellt.

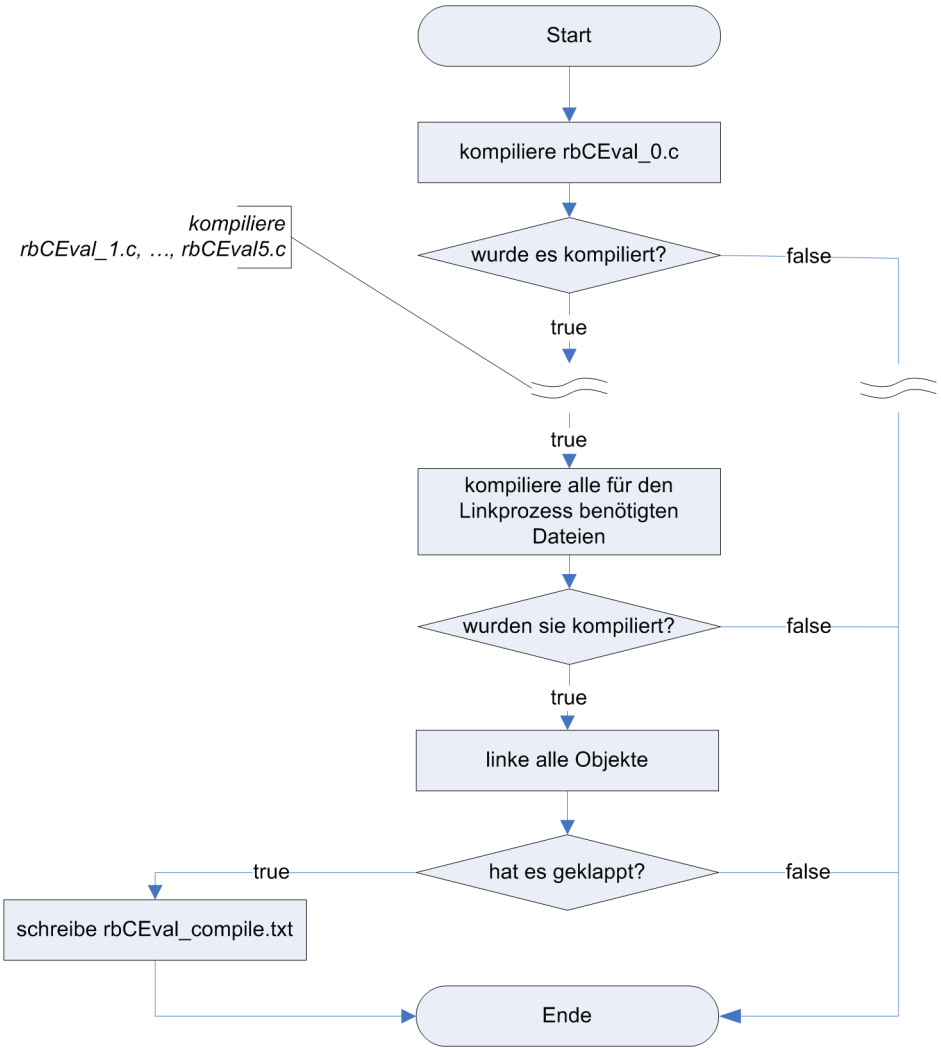


Abbildung 5.4: Build-Prozess, Ablaufdiagramm

5.3.2 Der Programmlauf

Über welche Schnittstelle der ausführbare Maschinencode heruntergeladen wird und wie die Ergebnisse des Programmlaufs vom Mikroprozessor gelesen werden, hängt von der verwendeten Hardware ab. Der Anwender muss dieses entweder manuell durchführen oder selbst automatisieren. Damit die Daten, die vom Mikroprozessor als Ergebnis gelesen werden, ein einheitliches Format haben, wird vorgegeben, wie die Funktion zum Übertragen/Speichern der Daten auszusehen hat und welche Funktionalität sie haben muss.
Eine einfach zu implementierende Möglichkeit, die auch die nötige Flexibilität für verschiedene Übertragungswege bietet, ist eine Funktion, die ein Zeichen, also ein Byte, speichert. Der Anwender muss dazu die Funktion mit dem folgenden Prototypen implementieren:

```
void rbCEval_outputByte(unsigned char);
```

Der generierte Quellcode greift, wie in Abschnitt 5.2.1.3 beschrieben, auf diese Funktion zu, um die Ergebnisse des Programmlaufs zu speichern.

Damit sämtliche Analysen durchgeführt werden, muss der Anwender sicherstellen, dass die folgende Funktion genau einmal aufgerufen wird:

```
1 void rbCEval_toolKit(void);
```

Diese Funktion wird vom Programmsystem bereitgestellt und ruft dann die Analysen auf. Dabei muss beachtet werden, dass die Analysen eine nicht definierte Laufzeit haben, so dass ein eventuell benutzter Watchdog nicht zeitgerecht bedient wird und den Prozessor neu starten würde.

5.3.3 Das Programmsystem

Der Build-Prozess muss so in das Programmsystem eingebunden werden, dass möglichst wenig Eingriffe vom Anwender nötig sind.
Der Programmablauf wird durch mehrere Perlskripte gesteuert, die nacheinander vom Anwender aufzurufen sind.

1. `init.pl`
   Überprüft den vom Anwender angepassten Build-Prozess und die Konfigurationsparameter, bevor die eigentlichen Analysen beginnen (siehe Abschnitt 5.3.3.2).
2. `run1.pl`
   Hauptskript, in dem alle Compilerfehleranalysen durchgeführt werden und der Maschinencode für den Hauptprogrammlauf erzeugt wird.
3. `run2.pl`
   Erzeugung des Maschinencodes für Modulo mit Null.
4. `run3.pl`
   Erzeugung des Maschinencodes für die Division durch Null.
5. `run4.pl`
   Erzeugung des Maschinencodes, um den Inhalt von Makros anzuzeigen.
6. `evaluation.pl`
   Wertet die Ergebnisse der Compilerfehleranalysen und der Programmläufe aus und gibt sie in lesbarer Form wieder.

Zusätzlich gibt es die folgenden Perlmodule:

- `rbCEval_config.pm`
  Konfigurationsdatei, in der der Anwender Verzeichnisse anpassen und das Kommando zum Starten des Build-Prozesses eintragen muss (siehe Abschnitt 5.3.3.2).
- `rbCEval_subroutines.pm`
  Enthält Subroutinen, die von mehreren oder allen Perlskripten genutzt werden.
- `rbCEval_vars.pm`
  Erstellt und initialisiert globale Variablen, die im gesamten Programmsystem verwendet werden.
- `rbCEval_const.pm`
  Enthält die Zuordnung der in Abschnitt 5.2.1.2 beschriebenen Integerkonstanten zu den entsprechenden Aussagen. Dieses Modul wird wie der Header `rbCEvalConst.h` automatisch erzeugt.

5.3.3.1 Kommunikation

Die Skripte sowie die Programmläufe arbeiten unabhängig voneinander. Damit die Ergebnisse den anderen Perlskripten und der Auswertung zur Verfügung stehen, werden diese in Textdateien zwischengespeichert. Jedes Skript schreibt seine Ergebnisse automatisch in eine Textdatei – `compResultx.txt` – wobei das `x` die Nummer des Skripts (`run1.pl - run4.pl`) darstellt. Der Anwender muss dafür sorgen , dass die Ergebnisse der Programmläufe jeweils in die Datein `run1.txt - run4.txt` geschrieben werden.
Unerwartete Fehler, die den Programmlauf auf dem Mikroprozessor unmöglich machen, werden in die Dateien `compResultx.txt` geschrieben. Diese Fehler müssten dann vom Anwender selbst oder vom Verantwortlichen des Programmsystems behoben werden. Solche Fehler sollten normalerweise nicht auftreten.

Das Programmsystem ist so ausgelegt, dass die Skripte jeweils auf die davor erzeugten Ergebnisse zugreifen können, um abhängig von deren Inhalt Code zu generieren, wie es auch im Datenmodell in Abbildung 4.2 dargestellt ist.
Dieser Datenfluss wäre nötig, wenn das Programmsystem mit einer beliebigen Größe des Datentyps `char` zurecht kommen müsste und es andere Darstellungen negativer ganzer Zahlen außer dem Zweierkomplement gäbe. Es wurde vereinbart, dass diese beiden Bedingungen vorausgesetzt werden, da es kaum noch Compiler oder Mikroprozessoren mit einer anderen Implementierung gibt. Es findet zwar eine Überprüfung der beiden Eigenschaften statt, die Ergebnisse fließen aber nicht in die anderen Analysen ein. Sollten die Bedingungen nicht erfüllt sein, wird in der Auswertung darauf hingewiesen und die anderen

Analysen für ungültig erklärt.
Dadurch wird ein Programmlauf eingespart und vereinfacht damit die Anwendung des Programmsystems

5.3.3.2 Konfiguration

Die vom Anwender durchzuführende Konfiguration des Programmsystems ist möglichst einfach gehalten. Sie ist gesondert in dem Perlmodul `rbCEval_config.pm` abgelegt, in der der Anwender die folgenden Einstellungen vornehmen muss:

1. Befehl zum Starten des Build-Prozesses.
2. Verzeichnistrennzeichen (`'\'` für Windows, `'/'` für Linux/Unix).
3. Zielpfad: Hier werden die generierten C-Dateien abgelegt (`rbCEval_x.c`).
4. Includepfad: Hier werden die Headerdateien abgelegt.
5. Temporärer Pfad: Hier werden alle Textdateien (`compResultx.txt`, `runx.txt`) abgelegt und der Build-Prozess muss hier die Datei `rbCEval_compile.txt` erzeugen, um einen erfolgreichen Durchlauf zu signalisieren.

Die geforderte Flexibilität wird durch den vom Anwender bereitzustellenden Build-Prozess erreicht. Das Programmsystem *CEval* wurde unter Windows 2000 entwickelt, läuft aber auch unter Unix bzw. Linux, wenn dort der entsprechende Perl Interpreter installiert ist (Version 5.8.4).

Dem Anwender wird die Möglichkeit gegeben, die Konfigurationsdatei gesondert abzulegen. Beim Aufruf der Perlskripte muss dann der entsprechende Pfad zu der Datei angegeben werden.[7]

5.3.3.3 Initialisierung

Die Konfiguration, die Anpassungen im Build-Prozess und die Implementierung der zusätzlichen C-Funktionen wird mit dem Skript `init.pl` geprüft. Es testet die folgenden drei Punkte:

1. Sind die Pfade korrekt?
2. Funktioniert die Compilerfehlererkennung mit der Textdatei?

[7] So kann die Konfigurationsdatei mit dem eigentlichen Projekt in ein Versionskontrollsystem wie CVS oder ClearCase abgelegt werden, so dass auch später noch die Analyse nachvollzogen werden kann

3. Lässt sich die Funktion `rbCEval_outputByte()` aufrufen?

Sollte einer der Punkte nicht erfüllt sein, wird eine entsprechende Fehlermeldung mit einem Hinweis auf mögliche Ursachen ausgegeben, die der Anwender dann überprüfen muss.

5.3.3.4 Generierung von Quellcode und Compilerfehleranalyse

Die Generierung des Quellcodes und die Compilerfehleranalyse geschieht in den Perlskripten `run1.pl - run4.pl`. Diese Skripte arbeiten nach dem gleichen Prinzip und unterscheiden sich nur in der Art und Menge der Durchläufe. Ein Durchlauf beinhaltet in diesem Zusammenhang die Codegenerierung, den Build-Prozess und die Compilerfehlerauswertung. Das Endergebnis jedes Skripts ist unter anderem der Maschinencode für einen Programmlauf auf dem Mikroprozessor. Die verschiedenen Durchläufe lassen sich in drei Kategorien unterteilen, die mit der Art der Analysen und der Erzeugung des Quellcodes zusammenhängt.

1. Für Analysen, bei denen gezielt Compilerfehler provoziert werden, wird der komplette Code dynamisch generiert. Dies geschieht zum Beispiel bei der Suche nach der maximalen Länge von Bezeichnern. Es werden solange Bezeichner erzeugt, bis ein Compilerfehler auftritt oder eine obere Grenze erreicht ist. Die erreichte Länge wird dann für die spätere Auswertung in eine Textdatei geschrieben. Ein anderes Beispiel ist die Frage, ob der Datentyp `long long` bekannt ist. Dazu wird ein Stück Code generiert, dass diesen Datentypen benutzt. Wird dieser Code ohne Fehler kompiliert, wird das Makro `RB_CEVAL_LONGLONG_DEFINED` gesetzt (vgl. Abschnitt 4.4.2), das dann im späteren Verlauf den entsprechenden Code aktiviert.

2. Vordefinierter Code wird getestet, bevor er für den eigentlichen Programmlauf zusammengefasst wird. Dazu wird jeweils der Inhalt einer Quelldatei in die Datei `rbCEval_0.c` kopiert[8] und die Funktion `rbCEval_toolKit()` in `rbCEval_4.c` generiert (siehe Beispielcode). In dieser Funktion werden dann die zu testenden Funktionen aufgerufen, so dass das Linken ebenfalls abgeprüft werden kann. Sollte hier ein Fehler auftreten, wird eine Fehlermeldung in die Datei `compResultx.txt` geschrieben und ausgegeben. In diesem Fall läuft der letzte Schritt nicht fehlerfrei und der vordefinierte Code muss überprüft werden.

[8] Den Inhalt zu kopieren ist notwendig, damit das Dateidatum aktualisiert wird. Beim Kopieren einer Datei wird dagegen das Datum nicht aktualisiert, was zu einer Fehlfunktion im Zusammenhang mit dem Tool Make führen könnte. Da Make das Datum und die Uhrzeit der letzten Änderung prüft (sekundengenau), wird nach jedem Durchlauf zusätzlich eine Sekunde gewartet, da die Generierung des Codes wesentlich schneller durchgeführt wird.

3. Im letzten Durchlauf wird der Code für den Programmlauf generiert. Dazu wird der Inhalt der getesteten Quellen zusammengefasst und anhand der Ergebnisse des ersten Schemas zusätzlicher Code generiert. Die Funktion `rbCEval_toolKit()` ist für diesen Durchlauf bereits vordefiniert und braucht nicht dynamisch erzeugt werden.

Die Durchläufe sind in den Skripten nach dem folgenden Schema umgesetzt:

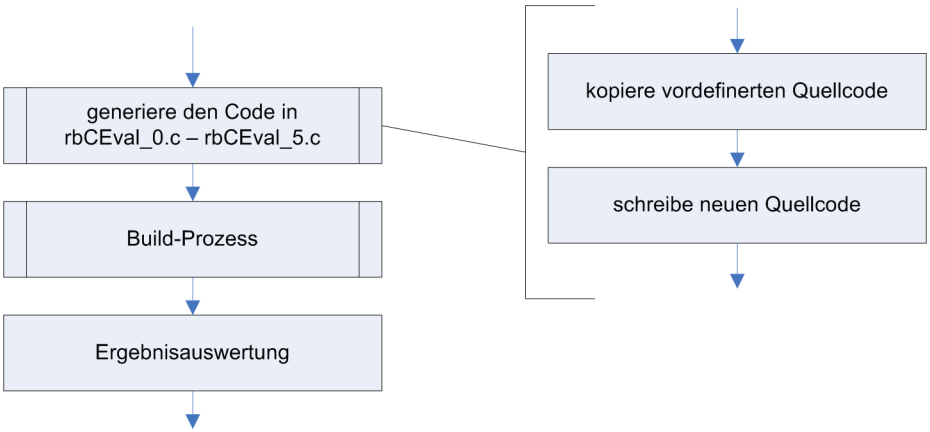


Abbildung 5.5: Schema einer Codegenerierung und Compilerfehleranalyse

Das Skript erzeugt die Dateien `rbCEval_x.c`, ruft den Build-Prozess auf, wertet dann dessen Ergebnis entsprechend aus und passt wenn nötig den Header `rbCEvalPredef.h` an.

Dieses Schema ist für jeden Durchlauf gleich. Der folgenden Codeausschnitt zeigt, wie dieser Ablauf in Perl umgesetzt wird:

```
1  # Perl Code
2
3  # Kopiere vordefinierten Quellcode
4  @sources = ( $sourcePath."rbCEval_001.c",
5               $sourcePath."dummy2.c",
6               $sourcePath."dummy3.c",
7               $sourcePath."rbCEvalTools.c",
8               $sourcePath."dummy_toolKit.c",
9               $sourcePath."rbCEvalOut.c" );
10 copySource(@sources, @targetFiles);
11
12 # schreibe neuen Quellcode
13 open FILE, ">".$targetFiles[4];
14 print FILE "#include \"rbCEval.h\" \n";
15 print FILE "void rbCEval_FILE(void){\n";
```

```
16 print FILE "  int result;\n";
17 print FILE "  rbCEval_sizeofByte(&result);\n";
18 print FILE "}\n";
19 close FILE;
20
21 # Starte den Build-Prozess
22 &compile($compileSkript);
23
24 # Werte das Ergebnis des Build-Prozesses aus
25 if(checkResult($compResult) == 1){
26   # Build-Prozess erfolgreich, evtl. Header anpassen
27   ...
28 }else {
29   ...
30 }
```

Zuerst wird in den Zeilen 4-9 das Array `@sources` mit Dateinamen angelegt, deren Inhalt mit Hilfe der Subroutine `copySource()` in Zeile 10 in die Dateien `rbCEval_x.c` geschrieben werden. Deren Dateinamen befinden sich im Array `@targetFiles`. Dieser Schritt muss durchgeführt werden, auch wenn kein vordefinierter Quellcode getestet werden soll. Für einen erfolgreichen Build-Prozess müssen die sechs Dateien `rbCEval_x.c` immer gültigen C-Code enthalten. In einer Datei muss die Funktion `rbCEval_toolKit()` definiert sein, da der Anwender diese aufrufen soll (siehe Abschnitt 5.3.2). Deshalb werden, wenn weniger als die sechs Dateien benötigt werden, Hilfsdateien genommen, in denen korrekter C-Code steht, der jedoch nichts ausführt.
Nachdem die Dateien `rbCEval_x.c` geschrieben wurden, wird der dynamische Code in einer der Dateien mit dem Hilfscode erzeugt. Der Hilfscode wird dabei überschrieben. In diesem Fall wird in Zeile 13 die Datei `rbCEval_4.c` neu geschrieben. In dieser wird die Funktion `rbCEval_toolKit()` erzeugt, welche die zu testende Funktion `rbCEval_sizeofByte()` aufruft, die in der Datei `rbCEval_001.c` vordefiniert ist (Zeile 4).
Damit ist der Quellcode für den Build-Prozess zusammengestellt und wird über die Funktion `compile()` in Zeile 22 gestartet. Diese Subroutine löscht vor dem Aufruf des Build-Prozesses die eventuell vorhandene Datei `rbCEval_compile.txt`.
Ob die Datei `rbCEval_compile.txt` nach dem Build-Prozess geschrieben wurde, um dessen Erfolg anzuzeigen, wird mit Hilfe der Subroutine `checkResult()` in Zeile 25 überprüft.

run1.pl Das Ablaufdiagramm in Abbildung 5.6 zeigt den Aufbau des Perlskripts, in dem die Compilerfehleranalysen durchgeführt werden und fast alle Analysen in einem Programmlauf zusammengefasst werden. Insgesamt wird in diesem Skript der Build-Prozess ca. 80-mal aufgerufen, bevor der endgültige Maschinencode erzeugt wird.

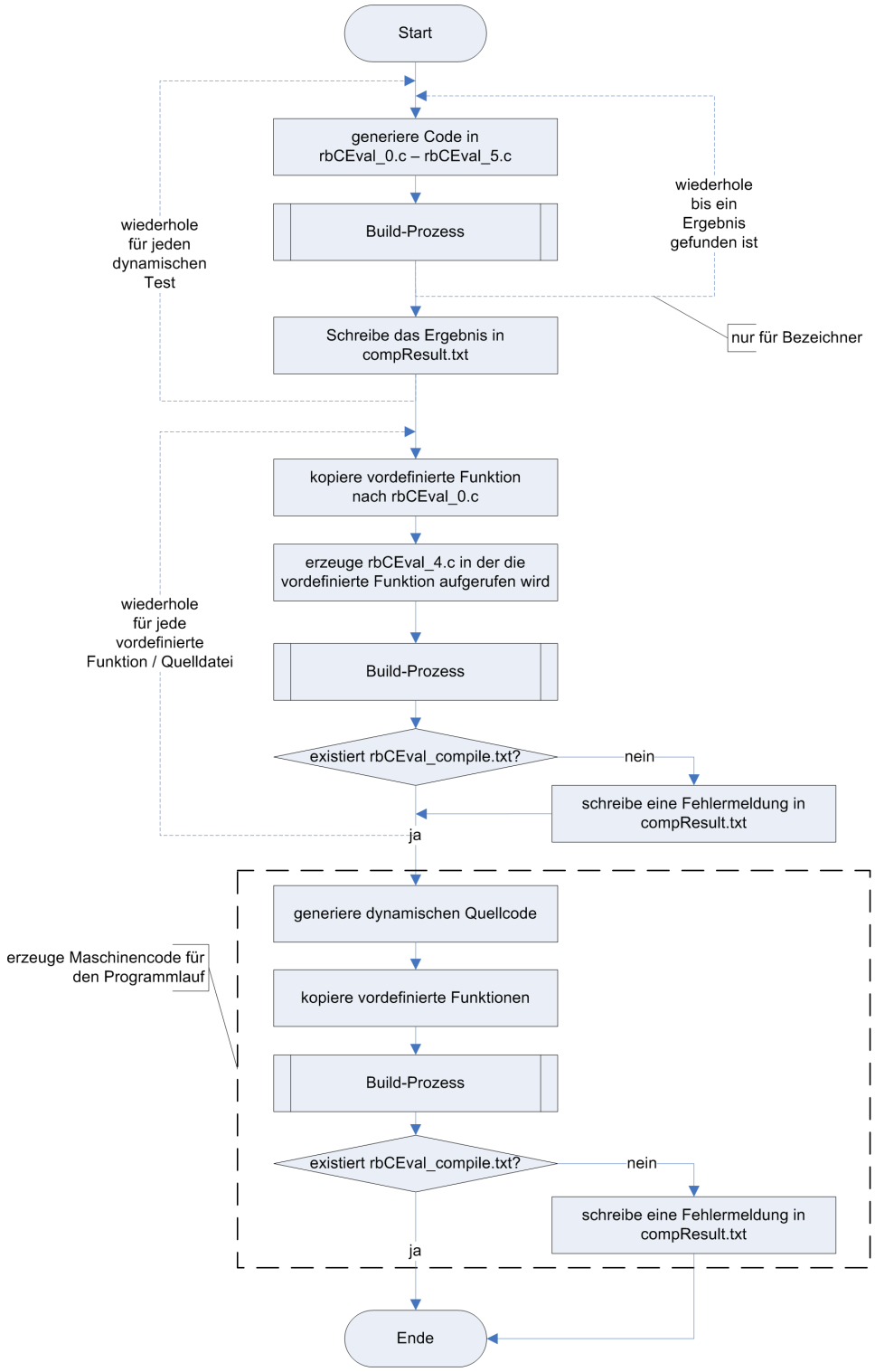


Abbildung 5.6: Ablaufdiagramm – run1.pl

run2.pl, run3.pl Die Untersuchungen von Modulo durch Null und der Division durch Null können zu Programmabstürzen führen. Deshalb müssen sie getrennt durchgeführt werden (`run2.pl`, `run3.pl`). So wird sicher gestellt, dass die anderen Analysen davon nicht betroffen sind.

run4.pl Die Analyse des Inhalts von Makros wird aus zwei Gründen separat in `run4.pl` durchgeführt:

1. Je nach Anzahl der zu untersuchenden Makros und deren Inhalt, vergrößert sich die Menge an Daten, die vom Mikroprozessor gelesen werden muss. Es kann sein, dass nicht beliebig viele Daten auf einmal übertragen werden können, weil sie beispielsweise im RAM oder in einem EEPROM zwischengespeichert werden. Da die Größe der Ergebnisse des ersten Programmlaufs schon bei 900 bis 1000 Bytes liegt, muss eine Vergrößerung der Datenmenge in diesem Durchlauf vermieden werden.
2. Es ist keine notwendige Analyse. Der Anwender kann entscheiden, ob und welche Makros er analysieren möchte.

5.3.3.5 Die Auswertung

Die Auswertung findet in dem Perlskript `evaluation.pl` statt und greift auf die Ergebnisse aus der Compilerfehleranalyse (`compResultx.txt`) und den verschiedenen Programmläufen (`runx.txt`) zurück. Dort sind die Ergebnisse im Format `name=wert` abgelegt (vgl. 5.2.1.3). Die Ergebnisse aus der Compilerfehleranalyse und den Programmläufen werden jeweils komplett in einen Hash[9] eingelesen, wobei der Name des Ergebnisses als Index genutzt wird. So können die Werte anhand des Namens ausgelesen und zugeordnet werden. Damit es keine Überschneidungen gibt, sind die Namen der Ergebnisse eindeutig gewählt. Der Wert stellt je nach Analyse entweder direkt das Ergebnis dar oder dient als Schlüssel (Konstante), mit dem das Ergebnis zugeordnet werden kann (vgl. 5.2.1.2). Als Unterscheidungsmerkmal werden die Namen der Ergebnisse, die einen Schlüssel liefern, in Großbuchstaben ausgegeben. Die Zuordnung zwischen Schlüssel und Erklärung ist in dem Perlmodul `rbCEval_const.pm` hinterlegt.
Die Ergebnisse werden mit erklärenden Beschreibungen ausgegeben. Zum Teil finden auch einfache Umrechnungen statt. Beispielsweise wird der Exponent von Gleitkommazahlen berechnet und die Größe der Datentypen von Byte in Bit umgerechnet. Das Endergebnis wird auf dem Bildschirm ausgegeben. In Kapitel 6 werden einige Ergebnisse miteinander verglichen, im Anhang C ist eines komplett dargestellt.

[9] Array, dass Strings als Index zulässt (Perl).

5.4 Review

Das Programmsystem wurde in Zusammenarbeit mit den Kollegen aus der Abteilung AE-BE/Eng3 entwickelt. Es fanden regelmäßige Rücksprachen hinsichtlich der Notwendigkeit von Analysen und des geplanten Ablaufs statt. Das Programmsystem wurde von Kollegen auf seine Anwendbarkeit getestet. Diese haben das Programmsystem ohne Einführung angewendet und die Analysen anhand des Benutzerhandbuchs durchgeführt. Anhand ihrer Vorschläge und Anmerkungen wurden das Programmsystem, dessen Benutzung, die Ausgabe der Ergebnisse und das Benutzerhandbuch optimiert.

5.5 Dokumentation

Die offizielle Firmensprache von Bosch ist Englisch. Deshalb sind das komplette Programm, die Kommentare im Quellcode und die Dokumentation in Englisch gehalten. Es gibt zwei verschiedene Dokumentationen.

Die eine richtet sich an den Anwender und beschreibt, wie das Programmsystem CEval zu verwenden ist. Sie gibt einen kurzen Einblick in die Arbeitsweise, eine Beschreibung der vom Anwender durchzuführenden Schritte und Erklärungen zu den Ergebnissen. Diese Anleitung ist in Word verfasst und im Anhang B hinterlegt.

Bei der zweiten handelt es sich um eine Administrator-Dokumentation. Hierin wird beschrieben, wie das Programm intern funktioniert, wie die verschiedenen Skripte und Dateien zusammengehören und wie das Programm erweitert werden kann. Diese Dokumentation ist im Quellcode integriert und lässt sich mit dem Tool Doxygen für die C-Dateien und mit dem Tool perldoc für die Perlskripte erzeugen. Beide Tools generieren HTML-Dateien.

Kapitel 6

Anwendung und Ergebnisse

In diesem Kapitel wird auf die Anwendung des Programmsystems und auf einige Ergebnisse eingegangen.
Während der Entwicklung wurde auf einem HC12 Mikroprozessor der Firma Freescale und direkt auf dem PC getestet. Für den HC12 wurde der Cosmic Compiler genutzt, für den PC der GNU Compiler. Um die Compiler besser vergleichen zu können, wurde für jeden Prozessor ein zweiter Compiler analysiert, der Metrowerks CodeWarrior der Firma Freescale für den HC12 und der freie lcc[1]-Compiler für den PC.

Zum Ende der Entwicklung wurde das Programmsystem in realen Projektumgebungen eingesetzt, auf einem ARM 32-Bit RISC Prozessor mit einem ARM Compiler und auf einem PowerPC mit einem GreenHills Compiler sowie mit dem GNU Compiler.

Es werden nun einige Ergebnisse vorgestellt und miteinander verglichen.
Die untersuchten Eigenschaften von Compilern desselben Prozessors unterscheiden sich häufig nur in wenigen Punkten. Diese lassen sich meist über Optionen anpassen. Gerade diese kleinen Unterschiede werden leicht übersehen. Werden verschiedene Prozessorarchitekturen verwendet, ist auch die Anzahl der Unterschiede größer.
Mit einigen Ergebnissen wurde so nicht gerechnet. Zum Beispiel wurden Enumerations von keinem der analysierten Compiler gleich umgesetzt, so dass die Enumerations zukünftig vorsichtiger verwendet werden. In einigen Fällen entsprachen die Eigenschaften nicht dem Standard.
Im Anhang C ist das Ergebnis des Cosmic Compilers mit dem HC12 Mikroprozessor vollständig dargestellt.

[1] `http://www.q-software-solutions.de/`

6.1 Gleicher Prozessor, verschiedene Compiler

6.1.1 HC12

Mikroprozessor:	MC9S12DP512 (HC12 16-Bit)
Compiler 1:	Cosmic HC(S)12 Compiler v4.6
Compiler 2:	Metrowerks CodeWarrior v4.6

Für die Analysen wurden beide Compiler mit ihren Standardeinstellungen verwendet. Die folgende Tabelle zeigt einige Unterschiede.

	Cosmic	**Metrowerks**
Länge von Bezeichnern	254	1023
plain char	unsigned	signed
sizeof(„wchar“)	16 Bit	8 Bit
sizeof(long long)	not defined	32 Bit
sizeof(double)	64 Bit	32 Bit
sizeof(long double)	64 Bit	32 Bit
Vorzeichen von plain int-Bitfeldern	unsigned	signed

Die Unterschiede bei den Enumerations zeigt die folgende. Es handelt sich dabei um einen Ausschnitt aus dem Vergleichstool „Beyond Compare“. Die Unterschiede sind rot gekennzeichnet :

Enumerations, size in bytes and signedness (1 = signe » d , 0 = unsigned) type			size	sign	sizeof(a »)	=	Enumerations, size in bytes and signedness (1 = signe » d , 0 = unsigned) type			size	sign	sizeof(a »)
{a,	b,	c}:	1	0	2	<>	{a,	b,	c}:	2	1	2
{a=-1,	b,	c}:	1	1	2		{a=-1,	b,	c}:	2	1	2
{a=128,	b,	c}:	1	0	2		{a=128,	b,	c}:	2	1	2
{a=-1,	b=128,	c}:	2	1	2	=	{a=-1,	b=128,	c}:	2	1	2
{a=256,	b,	c}:	2	0	2	<>	{a=256,	b,	c}:	2	1	2
{a=32768,	b,	c}:	2	0	2		{a=32768,	b,	c}:	not possible!		
{a=-1,	b= 32768,	c}:	4	1	4		{a=-1,	b= 32768,	c}:	not possible!		
{a=65536,	b,	c}:	4	0	4		{a=65536,	b,	c}:	not possible!		
{a=2147483648,	b,	c}:	4	1	4		{a=2147483648,	b,	c}:	not possible!		
{a=-1,	b=2147483648,	c}:	4	1	4		{a=-1,	b=2147483648,	c}:	not possible!		
{a=4294967296,	b,	c}: not possible!				=	{a=4294967296,	b,	c}: not possible!			

Beyond Compare 2.3.1

Abbildung 6.1: Enumarations auf dem HC12 Mikroprozessor (Cosmic - Metrowerks)

Wie sich erkennen lässt, legt der Metrowerks-Compiler standardmäßig für Enumerations ein `signed int` an, wohingegen der Cosmic Compiler die Größe den Werten der Enumeration anpasst.

Ein weiterer Unterschied ist die Implementierung von Gleitkommazahlen. Im Gegensatz zum Cosmic-Compiler werden beim Metrowerks Compiler alle Gleitkommazahlen gleich

gesetzt (Standardeinstellung). Abbildung 6.2 zeigt, dass auch die Verteilung der Bits eine andere ist. Hierbei handelt es sich allerdings um einen internen Fehler des Compilers. Da der gleiche Prozessor verwendet wird und somit die gleiche Gleitzkommaarithmetik, darf sich die Verteilung der Bits nicht ändern. Sie wird im Handbuch des Metrowerks-Compilers mit der Aufteilung Mantisse 23 Bits, Exponent 8 Bits und Vorzeichen 1 Bit angegeben [Fre06]. Daraufhin wurde der Algorithmus zur Analyse nochmals überprüft, jedoch kein Fehler gefunden. In der Analyse wird ausschließlich mit Gleitkommawerten gerechnet (siehe Codeausschnitt auf Seite 54). In einer Debugumgebung wurde jeder Schritt der Berechnung überprüft. Dabei stellte sich heraus, dass der Metrowerks Compiler, wenn nur mit Gleitkommazahlen gerechnet wird, bei Grenzwerten (Übergang zum nächsten Exponenten), fehlerhaft ist. Wird der Test mit Integerwerten durchgeführt, die dann implizit zu Gleitkommawerten konvertiert werden, tritt dieser Fehler nicht auf und das Ergebnis wäre korrekt. Dies ist aber nur für den Datentypen `float` möglich, da dessen Mantisse normalerweise kleiner ist als die Anzahl der Bits in einem `long`. Eine Analyse von `double` bzw. `long double` wäre dann nicht mehr möglich.[2]

Floating Point: Base = 2		=	Floating Point: Base = 2
precision (mantissa) of float:	23 bits	<>	precision (mantissa) of float: 22 bits
precision (mantissa) of double:	52 bits		precision (mantissa) of double: 22 bits
precision (mantissa) of long double:	52 bits		precision (mantissa) of long double: 22 bits
exponent of float:	8 bits		exponent of float: 9 bits
exponent of double:	11 bits		exponent of double: 9 bits
exponent of long double:	11 bits		exponent of long double: 9 bits
representable minimal difference (epsilon) to 1: (it might be smaller)		=	representable minimal difference (epsilon) to 1: (it might be smaller)
epsilon for float:	ca. 10^-6		epsilon for float: ca. 10^-6
epsilon for double:	ca. 10^-15	<>	epsilon for double: ca. 10^-6
epsilon for long double:	ca. 10^-15		epsilon for long double: ca. 10^-6

Beyond Compare 2.3.1

Abbildung 6.2: Vergleich von Gleitkommazahlen auf dem HC12 Mikroprozessor

Im Anhang C befindet sich ein vollständiger Vergleich der beiden Ergebnisse.

[2]Die Mantisse von float hat normalerweise 23 Bit, die Anzahl der Bits in einem long ist mindestens 32. In einem float lassen sich Integerwerte bis zu 2^{24} ohne Genauigkeitsverlust darstellen, in einem double mit normalerweise 52 Mantissenbits bis zu 2^{53}. Dieser Wert ist als Integerwert in einem long mit 32 Bit nicht mehr darstellbar.

6.1.2 PowerPC

Mikroprozessor:	MPC5516G (PowerPC, 32 Bit)
Compiler 1:	GreenHills Multi v5.0.1
Compiler 2:	GNU C Compiler v3.4.4

Auf dem PowerPC gibt es bei den analysierten Compilern fast keine Unterschiede. Neben kleineren Unterschieden bei den Enumerations wird nur das `int`-Bitfeld anders implementiert. Der GCC interpretiert ein einfaches `int` bei Bitfeldern als `signed int` wohingegen der GreenHills es als `unsigned int` ansieht.

6.2 Unterschiedliche Prozessoren, verschiedene Compiler

Mikroprozessor 1:	MPC5516G (PowerPC, 32 Bit)
Compiler 1:	GreenHills Multi v5.0.1
Mikroprozessor 2:	MC9S12DP512 (HC12 16-Bit)
Compiler 2:	Cosmic HC(S)12 Compiler v4.6

Bei diesem Vergleich gibt es größere Unterschiede. Gerade die Größe und das Alignment der Datentypen unterscheidet sich stark, da es sich bei dem HC12 um einen 16-Bit Prozessor und bei dem PowerPC um einen 32-Bit Prozessor handelt.

	Cosmic (HC12)	**GreenHills (PowerPC)**
Länge von Bezeichnern	254	1023
Größe(Alignment):		
char	8(1)	8(1)
wchar	16	32
short	16(1)	16(2)
int	16(1)	32(4)
long	32(1)	32(4)
longlong	not defined	64(8)
float	32(1)	32(4)
double	64(1)	64(8)
longdouble	64(1)	64(8)
sizeof	16	32
pointer to data	16(1)	32(4)
pointerDiff	16	32
pointer to function	16(1)	32(4)
Enumerations	sieh Abb. 6.1 links	immer signed int
Füllbytes in Strukturen	Nein	Ja

Neben der Datentypgröße gibt es noch weitere Unterschiede, besonders bei der Implementierung von Bitfeldern, wie in Abbildung 6.3 zu sehen ist. Es zeigt sich, dass bei diesen Einstellungen keine Eigenschaft der Bitfelder genau übereinstimmt (vgl. 3.1.9). Der Cosmic Compiler weicht hier vom Standard ab, da Bitfelder, egal ob mit `signed` oder `unsigned` deklariert, immer als `unsigned` interpretiert werden.

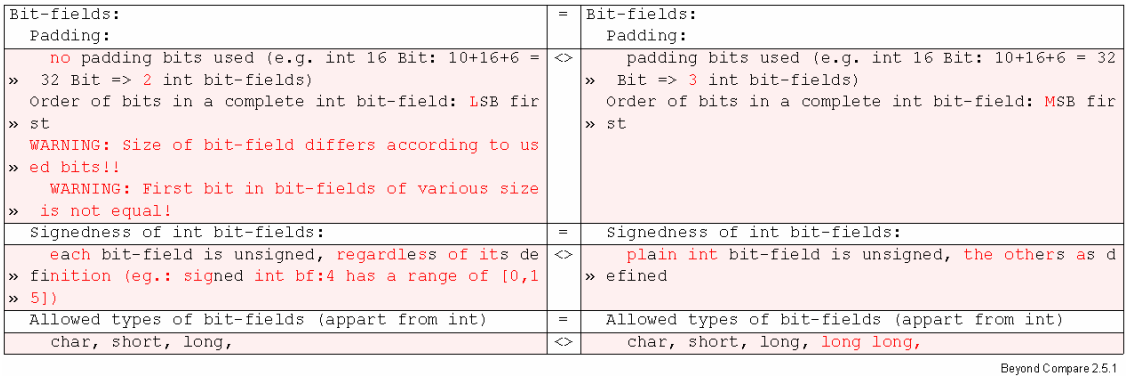

Bit-fields:
Padding:
=
Bit-fields:
Padding:
no padding bits used (e.g. int 16 Bit: 10+16+6 = 32 Bit => 2 int bit-fields)
<>
padding bits used (e.g. int 16 Bit: 10+16+6 = 32 Bit => 3 int bit-fields)
Order of bits in a complete int bit-field: LSB first
Order of bits in a complete int bit-field: MSB first
WARNING: Size of bit-field differs according to used bits!!
WARNING: First bit in bit-fields of various size is not equal!
Signedness of int bit-fields:
=
Signedness of int bit-fields:
each bit-field is unsigned, regardless of its definition (eg.: signed int bf:4 has a range of [0,15])
<>
plain int bit-field is unsigned, the others as defined
Allowed types of bit-fields (appart from int)
=
Allowed types of bit-fields (appart from int)
char, short, long,
<>
char, short, long, long long,
Beyond Compare 2.5.1

Abbildung 6.3: Vergleich von Bitfeldern beim HC12(Cosmic) und PowerPC(GreenHills)

6.3 Unterschiedliche Prozessoren, gleicher Compiler

Mikroprozessor 1:	MPC5516G (PowerPC, 32 Bit)
Mikroprozessor 2:	Pentium 4 (Intel, 32 Bit)
Compiler:	GNU C Compiler v3.4.4

Bei diesem Test wurde der GNU Compiler für den PowerPC als Crosscompiler verwendet. Da es sich um den gleichen Compiler handelt, hängen die wenigen Unterschiede mit dem Mikroprozessor zusammen.

	PowerPC	**Intel**
Größe(Alignment):		
wchar	32	16
longdouble	64(8)	96(4)
Endianess	Big-Endian	Little-Endian
Division durch 0 (Laufzeit)	= Dividend	Laufzeitfehler
Modulo durch 0 (Laufzeit)	undefiniert	Laufzeitfehler
Bitfelder:		
Erstes Bit	MSB	LSB

Hier kann ein Unterschied zwischen eingebetteten und „normalen“ Systemen festgestellt werden. Modulo und Division durch Null führt auf dem PC zum Programmabsturz. Dies ist nicht kritisch, da das Betriebsystem dafür sorgt, dass das abgestürzte Programm beendet wird oder dem Benutzer eine Fehlermeldung anzeigt. Auf dem PowerPC ergibt Modulo durch null zwar kein definiertes Resultat, das Programm läuft aber weiter.

Zusammenfassung

Die Aufgabe bestand darin, ein Programm zu entwickeln, dass die Implementierung von nicht eindeutig definierten Eigenschaften der Programmiersprache C bei Compilern für eingebettete Systeme untersucht. Dies sollte auf Basis des Standards C90 geschehen. Hintergrund der Aufgabe ist der Einsatz von verschiedenen Compilern und Mikroprozessoren bei der Entwicklung von Steuergeräten im Automotive-Bereich. Wesentlich war daher die Anwendbarkeit des Programmsystems auf beliebige Compiler-Prozessor-Kombinationen sowie die Übertragbarkeit der Ergebnisse auf reale Anwenderprojekte.
Die Analyse des Standards C90 zeigte, dass er eine Fülle von Eigenschaften enthält, die nicht definiert sind. Es wurde deshalb hauptsächlich auf die „implementation defined features“ eingegangen. Deren Untersuchung ergab, dass nur ein Teil dieser Eigenschaften für eingebettete Systeme relevant sind und deshalb nur diese analysiert werden müssen.
Die Analysen der Eigenschaften sind in ein einfaches Programmsystem integriert worden, das für den Einsatz in bestimmte Projekte angepasst werden kann. Die nötige Flexibilität wird durch die Einbindung des Anwenders sichergestellt, der über den Build-Prozess die Schnittstelle zum jeweiligen Compiler zur Verfügung stellen muss.
Zur Anwendung dieses Programmsystems wird deshalb eine gewisse Erfahrung mit der Programmierung von Mikroprozessoren in C vorausgesetzt. Ferner muss der zu analysierende Compiler konform zum Standard C90 sein. Das Programmsystem testet nur bestimmte Eigenschaften, die der Standard offen lässt und nicht, ob der Standard von dem Compiler erfüllt wird. Ist der Compiler nicht C90-konform, kann es prinzipiell zu Fehlern in der Programmausführung und den Ergebnissen kommen.
Die Anwendung des Programms auf verschiedene Compiler hat gezeigt, dass es den Anforderungen entspricht und wichtige Unterschiede zwischen den Compilern offenlegt. Die Ergebnisse geben dem Anwender Hinweise auf Probleme bei der Verwendung von bestimmten Sprachelementen der Programmiersprache C.
Es war nicht möglich, für alle relevanten Eigenschaften Analysen zu entwerfen. Erstens, weil die Analysen erheblichen Mehraufwand für den Anwender bedeutet hätten, zweitens, weil nicht genug Zeit zur Verfügung stand und drittens, weil die Übertragbarkeit der Ergebnisse der Analysen nicht gegeben war.

Ausblick

Das Programmsystem *CEval* bietet eine Basis für weitere Analysen, die sich nicht nur auf den Standard C90 beziehen müssen. Durch den modularen Aufbau in mehreren Perlskripten lassen sich neue Analysen ohne großen Aufwand einbinden.
Vorstellbar sind unter anderem Analysen zu den folgenden Themen:

- Untersuchung der undefinierten und nicht näher spezifizierten Eigenschaften („undefined and unspecified features“)
- Überprüfung, ob der Compiler bestimmte Eigenschaften gemäß dem Standard implementiert hat:
  Die bisherigen Analysen haben gezeigt, dass einige Compiler sich nicht immer genau an den Standard halten, beispielsweise werden als `signed` deklarierte Bitfelder einfach als `unsigned` interpretiert. Daher ist es sinnvoll, auch definierte Eigenschaften zu untersuchen, wenn sie in Verdacht stehen, vom Standard abzuweichen.
- Laufzeitmessungen:
  Mit Laufzeitmessungen könnte die Effizienz des Compilers bei der Umsetzung bestimmter Programmkonstrukte untersucht werden. Beispielsweise, wie schnell Berechnungen mit Gleitkommazahlen durchgeführt werden oder, wie effizient Schleifen umgesetzt werden. Hierfür müssten Timer genutzt werden. Diese können nicht allgemein angesprochen werden und müssten vom Anwender implementiert werden. Ist eine Laufzeitmessung einmal implementiert, ließe sich mit Hilfe dieser ein Benchmark entwickeln, mit dem sich die Effizienz verschiedener Compiler-Prozessor-Kombinationen vergleichen ließe.
- Atomare Operationen:
  Ob beispielsweise die Zuweisung `a = b;` atomar ist, hängt davon ab, wie der entsprechende Systembefehl implementiert ist. Können nur 8-Bit Werte direkt verschoben werden, ist diese Zuweisung nur bei `char`-Variablen atomar. Untersucht werden könnte dieses Verhalten mit Hilfe von Interrupts, die vom Anwender implementiert werden müssten. Zuverlässig könnte jedoch nur bestimmt werden, ob ein Operator nicht atomar ist.
- Untersuchung des Stack-Bedarfs zur Laufzeit:
  Es könnte untersucht werden, wie und wo automatische Variablen im Speicher abgelegt werden und wie groß der resultierende Speicherverbrauch ist. Damit kann beispielsweise untersucht werden, ob nur der Platz belegt wird, der für die abzulegenden Daten benötigt wird. Dabei ist zu beachten, dass automatische Variablen auch in Registern abgelegt werden können.

- Untersuchung von „Inlining“:
  „Inlining“ bezeichnet das „Kopieren“ des Inhalts einer Funktion an die Stelle, an der die Funktion aufgerufen wird. Es wird zur Laufzeitoptimierung verwendet, da dadurch ein Funktionsaufruf (Sprung im Maschinencode) vermieden wird. Es könnte untersucht werden, ob durch „Inlining“ tatsächlich eine Optimierung erreicht wird.

- Eigenschaften des Standards C99:
  Bei den Untersuchungen ist deutlich geworden, dass viele Compiler schon Eigenschaften des neueren Standards C99 integriert haben. Compiler könnten deshalb gezielt auf bestimmte neue Eigenschaften untersucht werden. Für den Datentyp `long long` und dem `'//'`-Kommentar sind bereits Analysen implementiert.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen Quellen und Hilfsmittel als die aufgeführten verwendet habe. Die benutzten Quellen, wörtlich oder inhaltlich entnommenen Stellen habe ich als solche kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift

Glossar

Alignment Die Zahl, durch die die Speicheradresse teilbar sein muss.

ANSI American National Standardization Institute.

Balancing Angleichen von Datentypen bei arithmetischen Operationen.

Big-Endian Das höchstwertigste Byte einer Variable/Zahl liegt auf der niederwertigsten Speicheraddresse.

C90 ISO/IEC 9899:1990, C-Standard.

C99 ISO/IEC 9899:1999, C-Standard, neuere Version.

Cast Datentypkonvertierung.

Compiler übersetzt den Quellcode in Maschinensprache, erzeugt Objektcode.

Einerkomplement Eine Darstellung negativer Binärzahlen. Der Betrag einer negativen Zahl ergibt sich, wenn alle Bits invertiert werden, hat zwei Darstellungen für 0.

EN 29899 Europäische Norm, entspricht inhaltlich C90.

Endianess Anordnung der Bytes einer Variable.

IEC International Electrotechnical Commission.

Integer Promotion Alle Zahlen und Datentypen kleiner int werden bei Berechnungen zu int konvertiert.

ISO Internatinal Organization for Standardization.

Linker Löst Abhängigkeiten zwischen Modulen auf und verbindet den Objektcode zum fertigen Programm (siehe Compiler).

Little-Endian Das niederwertigste Byte einer Variable/Zahl liegt auf der niederwertigsten Speicherdaddresse.

Middle-Endian Mischform von Little- und Big-Endian.

Mixed-Endian Siehe Middle-Endian.

Padding Füllbytes und Bits, damit das Alignment gewährleistet werden kann.

Stringize Operator Präprozessor-Operator, der eine beliebige Zeichenkette während des 'Preprocessing' in einen String umwandelt (#Kette → "Kette").

Zweierkomplement Eine Darstellung negativer Binärzahlen. Der Betrag einer negativen Zahl ergibt sich, wenn alle Bits invertiert werden und 1 hinzuaddiert wird.

Literaturverzeichnis

[Cos05] Cosmic Software: *C Cross Compiler Users Guide for Freescale HC12/HCS12*, 2005. Version 4.6.

[Cos07] Cosmic Software: *Cosmic Software C Compilers*, 2007. [Online; Stand 30.05.2007; `http://www.cosmic-software.com/compiler.php`].

[Fre06] Freescale Semiconductor, Inc: *HC(S) Compiler Manual*, 2006.

[GCC07] GCC Team: *Status of C99 features in GCC*, 2007. [Online; Stand 13.02.2007; `http://gcc.gnu.org/c99status.html`].

[Gre07] Green Hills Software Inc.: *Green Hills Optimizing C Compilers*, 2007. [Online; Stand 30.05.2007; `http://www.ghs.com/products/c_optimizing_compilers.html`].

[ISO90] ISO/IEC: *Programming languages – C*. International Standard 9899, International Organization of Standardization, 1990. [DIN EN 29899].

[ISO99] ISO/IEC: *Programming languages – C*. International Standard 9899, International Organization of Standardization, 1999.

[MIS04] MISRA: *MISRA-C:2004 – Guidelines for the use of the C language in critical systems*. Richtlinie, The Motor Industry Software Reliability Association, 2004.

[Tex03] Texas Instruments: *TMS320C55x Optimizing C/C++ Compiler Users Guide*, 2003. [Online; Stand 14.08.2007; `http://focus.ti.com/lit/ug/spru281f/spru281f.pdf`].

[Wik07a] Wikipedia: *Endianness — Wikipedia, The Free Encyclopedia*, 2007. [Online; Stand 31.07.2007; `http://en.wikipedia.org/w/index.php?title=Endianness&oldid=147690577`].

[Wik07b] Wikipedia: *Varianten der Programmiersprache C — Wikipedia, Die freie Enzyklopädie*, 2007. [Online; Stand 30.05.2007; `http://de.wikipedia.org/w/index.php?title=Varianten_der_Programmiersprache_C&oldid=31094790`].

Abbildungsverzeichnis

2.1 Konfigurationsmenü von QA-C 9

3.1 Rechtsverschiebung bei signed Werten 16
3.2 Anordnung von Bytes 18
3.3 Alignment in Strukturen 20
3.4 Padding in Bitfeldern 20
3.5 Anordnung der Bits 21
3.6 Verschiebung des LSB bei Big-Endian 21

4.1 Datenflussdiagramm - einfach 30
4.2 Datenflussdiagramm - erweitert 31

5.1 Füllbyte am Ende von Strukturen 50
5.2 Binäre Suche der Bezeichnerlänge 58
5.3 Gesamtablauf 62
5.4 Build-Prozess, Ablaufdiagramm 64
5.5 Schema einer Codegenerierung und Compilerfehleranalyse 69
5.6 Ablaufdiagramm – run1.pl 71

6.1 Enumarations auf dem HC12 Mikroprozessor (Cosmic - Metrowerks) 75
6.2 Vergleich von Gleitkommazahlen auf dem HC12 Mikroprozessor 76
6.3 Vergleich von Bitfeldern beim HC12(Cosmic) und PowerPC(GreenHills) . . 78

Anhang A

Portability Issues

Anhang G des Standards ISO/IEC 9899:1990
Der deutschen Norm aus dem Jahr 1994 entnommen: DIN EN 29899:1994

Wiedergegeben mit Erlaubnis des DIN Deutsches Institut für Normung e.V. Maßgebend für das Anwenden der DIN-Norm ist deren Fassung mit dem neusten Ausgabedatum, die bei der Beuth Verlag GmbH, Burggrafenstraße 6, 10787 Berlin, erhältlich ist.

Annex G
(informative)
Portability issues

This annex collects some information about portability that appears in this International Standard.

G.1 Unspecified behavior

The following are unspecified:

— The manner and timing of static initialization (5.1.2).

— The behavior if a printable character is written when the active position is at the final position of a line (5.2.2).

— The behavior if a backspace character is written when the active position is at the initial position of a line (5.2.2).

— The behavior if a horizontal tab character is written when the active position is at or past the last defined horizontal tabulation position (5.2.2).

— The behavior if a vertical tab character is written when the active position is at or past the last defined vertical tabulation position (5.2.2).

— The representations of floating types (6.1.2.5).

— The order in which expressions are evaluated — in any order conforming to the precedence rules, even in the presence of parentheses (6.3).

— The order in which side effects take place (6.3).

— The order in which the function designator and the arguments in a function call are evaluated (6.3.2.2).

— The alignment of the addressable storage unit allocated to hold a bit-field (6.5.2.1).

— The layout of storage for parameters (6.7.1).

— The order in which **#** and **##** operations are evaluated during macro substitution (6.8.3.3).

— Whether **errno** is a macro or an external identifier (7.1.4).

— Whether **setjmp** is a macro or an external identifier (7.6.1.1).

— Whether **va_end** is a macro or an external identifier (7.8.1.3).

— The value of the file position indicator after a successful call to the **ungetc** function for a text stream, until all pushed-back characters are read or discarded (7.9.7.11)..

— The details of the value stored by the **fgetpos** function on success (7.9.9.1).

— The details of the value returned by the **ftell** function for a text stream on success (7.9.9.4).

— The order and contiguity of storage allocated by the **calloc**, **malloc**, and **realloc** functions (7.10.3).

— Which of two elements that compare as equal is returned by the **bsearch** function (7.10.5.1).

— The order in an array sorted by the **qsort** function of two elements that compare as equal (7.10.5.2).

— The encoding of the calendar time returned by the **time** function (7.12.2.3).

G.2 Undefined behavior

The behavior in the following circumstances is undefined:

— A nonempty source file does not end in a new-line character, ends in new-line character immediately preceded by a backslash character, or ends in a partial preprocessing token or comment (5.1.1.2).

— A character not in the required character set is encountered in a source file, except in a preprocessing token that is never converted to a token, a character constant, a string literal, a header name, or a comment (5.2.1).

— A comment, string literal, character constant, or header name contains an invalid multibyte character or does not begin and end in the initial shift state (5.2.1.2).

— An unmatched ' or " character is encountered on a logical source line during tokenization (6.1).

— The same identifier is used more than once as a label in the same function (6.1.2.1).

— An identifier is used that is not visible in the current scope (6.1.2.1).

— Identifiers that are intended to denote the same entity differ in a character beyond the minimal significant characters (6.1.2).

— The same identifier has both internal and external linkage in the same translation unit (6.1.2.2).

— The value stored in a pointer that referred to an object with automatic storage duration is used (6.1.2.4).

— Two declarations of the same object or function specify types that are not compatible (6.1.2.6).

— An unspecified escape sequence is encountered in a character constant or a string literal (6.1.3.4).

— An attempt is made to modify a string literal of either form (6.1.4).

— A character string literal token is adjacent to a wide string literal token (6.1.4).

— The characters ', \, ", or /* are encountered between the < and > delimiters or the characters ', \, or /* are encountered between the " delimiters in the two forms of a header name preprocessing token (6.1.7).

— An arithmetic conversion produces a result that cannot be represented in the space provided (6.2.1).

— An lvalue with an incomplete type is used in a context that requires the value of the designated object (6.2.2.1).

— The value of a void expression is used or an implicit conversion (except to **void**) is applied to a void expression (6.2.2.2).

— An object is modified more than once, or is modified and accessed other than to determine the new value, between two sequence points (6.3).

— An arithmetic operation is invalid (such as division or modulus by 0) or produces a result that cannot be represented in the space provided (such as overflow or underflow) (6.3).

— An object has its stored value accessed by an lvalue that does not have one of the following types: the declared type of the object, a qualified version of the declared type of the object, the signed or unsigned type corresponding to the declared type of the object, the signed or unsigned type corresponding to a qualified version of the declared type of the object, an aggregate or union type that (recursively) includes one of the aforementioned types among its members, or a character type (6.3).

— An argument to a function is a void expression (6.3.2.2).

— For a function call without a function prototype, the number of arguments does not agree with the number of parameters (6.3.2.2).

— For a function call without a function prototype, if the function is defined without a function prototype, and the types of the arguments after promotion do not agree with those of the parameters after promotion (6.3.2.2).

— If a function is called with a function prototype and the function is not defined with a compatible type (6.3.2.2).

— A function that accepts a variable number of arguments is called without a function prototype that ends with an ellipsis (6.3.2.2).

— An invalid array reference, null pointer reference, or reference to an object declared with automatic storage duration in a terminated block occurs (6.3.3.2).

— A pointer to a function is converted to point to a function of a different type and used to call a function of a type not compatible with the original type (6.3.4).

— A pointer to a function is converted to a pointer to an object or a pointer to an object is converted to a pointer to a function (6.3.4).

— A pointer is converted to other than an integral or pointer type (6.3.4).

— A pointer that does not behave like a pointer to an element of an array object is added to or subtracted from (6.3.6).

— Pointers that do not behave as if they point to the same array object are subtracted (6.3.6).

— An expression is shifted by a negative number or by an amount greater than or equal to the width in bits of the expression being shifted (6.3.7).

— Pointers are compared using a relational operator that do not point to the same aggregate or union (6.3.8).

— An object is assigned to an overlapping object (6.3.16.1).

— An identifier for an object is declared with no linkage and the type of the object is incomplete after its declarator, or after its init-declarator if it has an initializer (6.5).

— A function is declared at block scope with a storage-class specifier other than **`extern`** (6.5.1).

— A structure or union is defined as containing only unnamed members (6.5.2.1).

— A bit-field is declared with a type other than **`int`**, **`signed int`**, or **`unsigned int`** (6.5.2.1).

— An attempt is made to modify an object with const-qualified type by means of an lvalue with non-const-qualified type (6.5.3).

— An attempt is made to refer to an object with volatile-qualified type by means of an lvalue with non-volatile-qualified type (6.5.3).

— The value of an uninitialized object that has automatic storage duration is used before a value is assigned (6.5.7).

— An object with aggregate or union type with static storage duration has a non-brace-enclosed initializer, or an object with aggregate or union type with automatic storage duration has either a single expression initializer with a type other than that of the object or a non-brace-enclosed initializer (6.5.7).

— The value of a function is used, but no value was returned (6.6.6.4).

— An identifier with external linkage is used but there does not exist exactly one external definition in the program for the identifier (6.7).

— A function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation (6.7.1).

— An identifier for an object with internal linkage and an incomplete type is declared with a tentative definition (6.7.2).

— The token **defined** is generated during the expansion of a **#if** or **#elif** preprocessing directive (6.8.1).

— The **#include** preprocessing directive that results after expansion does not match one of the two header name forms (6.8.2).

— A macro argument consists of no preprocessing tokens (6.8.3).

— There are sequences of preprocessing tokens within the list of macro arguments that would otherwise act as preprocessing directive lines (6.8.3).

— The result of the preprocessing operator # is not a valid character string literal (6.8.3.2).

— The result of the preprocessing concatenation operator ## is not a valid preprocessing token (6.8.3.3).

— The **#line** preprocessing directive that results after expansion does not match one of the two well-defined forms (6.8.4).

— One of the following identifiers is the subject of a **#define** or **#undef** preprocessing directive: **defined**, **__LINE__**, **__FILE__**, **__DATE__**, **__TIME__**, or **__STDC__** (6.8.8).

— An attempt is made to copy an object to an overlapping object by use of a library function other than **memmove** (clause 7).

— The effect if a standard header is included within an external definition; is included for the first time after the first reference to any of the functions or objects it declares, or to any of the types or macros it defines; or is included while a macro is defined with a name the same as a keyword (7.1.2).

— The effect if the program redefines a reserved external identifier (7.1.3).

— A macro definition of **errno** is suppressed to obtain access to an actual object (7.1.4).

— The parameter *member-designator* of an **offsetof** macro is an invalid right operand of the operator for the *type* parameter or designates bit-field member of a structure (7.1.6).

— A library function argument has an invalid value, unless the behavior is specified explicitly (7.1.7).

— A library function that accepts a variable number of arguments is not declared (7.1.7).

— The macro definition of **assert** is suppressed to obtain access to an actual function (7.2).

— The argument to a character handling function is out of the domain (7.3).

— A macro definition of **setjmp** is suppressed to obtain access to an actual function (7.6).

— An invocation of the **setjmp** macro occurs in a context other than as the controlling expression in a selection or iteration statement, or in a comparison with an integral constant expression (possibly as implied by the unary ! operator) as the controlling expression of a selection or iteration statement, or as an expression statement (possibly cast to **void**) (7.6.1.1).

— An object of automatic storage class that does not have volatile-qualified type has been changed between a **setjmp** invocation and a **longjmp** call and then has its value accessed (7.6.2.1).

— The **longjmp** function is invoked from a nested signal routine (7.6.2.1).

— A signal occurs other than as the result of calling the **abort** or **raise** function, and the signal handler calls any function in the standard library other than the **signal** function itself or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type **volatile sig_atomic_t** (7.7.1.1).

— The value of **errno** is referred to after a signal occurs other than as the result of calling the **abort** or **raise** function and the corresponding signal handler calls the **signal** function such that it returns the value **SIG_ERR** (7.7.1.1).

— The macro **va_arg** is invoked with the parameter **ap** that was passed to a function that invoked the macro **va_arg** with the same parameter (7.8).

— A macro definition of **va_start**, **va_arg**, or **va_end** or a combination thereof is suppressed to obtain access to an actual function (7.8.1).

— The parameter *parmN* of a **va_start** macro is declared with the **register** storage class, or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions (7.8.1.1).

— There is no actual next argument for a **va_arg** macro invocation (7.8.1.2).

— The type of the actual next argument in a variable argument list disagrees with the type specified by the **va_arg** macro (7.8.1.2).

— The **va_end** macro is invoked without a corresponding invocation of the **va_start** macro (7.8.1.3).

— A return occurs from a function with a variable argument list initialized by the **va_start** macro before the **va_end** macro is invoked (7.8.1.3).

— The stream for the **fflush** function points to an input stream or to an update stream in which the most recent operation was input (7.9.5.2).

— An output operation on an update stream is followed by an input operation without an intervening call to the **fflush** function or a file positioning function, or an input operation on an update stream is followed by an output operation without an intervening call to a file positioning function (7.9.5.3).

— The format for the **fprintf** or **fscanf** function does not match the argument list (7.9.6).

— An invalid conversion specification is found in the format for the **fprintf** or **fscanf** function (7.9.6).

— A **%%** conversion specification for the **fprintf** or **fscanf** function contains characters between the pair of **%** characters (7.9.6).

— A conversion specification for the **fprintf** function contains an **h** or **l** with a conversion specifier other than **d**, **i**, **n**, **o**, **u**, **x**, or **X**, or an **L** with a conversion specifier other than **e**, **E**, **f**, **g**, or **G** (7.9.6.1).

— A conversion specification for the **fprintf** function contains a **#** flag with a conversion specifier other than **o**, **x**, **X**, **e**, **E**, **f**, **g**, or **G** (7.9.6.1).

— A conversion specification for the **fprintf** function contains a **0** flag with a conversion specifier other than **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, or **G** (7.9.6.1).

— An aggregate or union, or a pointer to an aggregate or union is an argument to the **fprintf** function, except for the conversion specifiers **%s** (for an array of character type) or **%p** (for a pointer to **void**) (7.9.6.1).

— A single conversion by the **fprintf** function produces more than 509 characters of output (7.9.6.1).

— A conversion specification for the **fscanf** function contains an **h** or **l** with a conversion specifier other than **d**, **i**, **n**, **o**, **u**, or **x**, or an **L** with a conversion specifier other than **e**, **f**, or **g** (7.9.6.2).

— A pointer value printed by **%p** conversion by the **fprintf** function during a previous program execution is the argument for **%p** conversion by the **fscanf** function (7.9.6.2).

— The result of a conversion by the **fscanf** function cannot be represented in the space provided, or the receiving object does not have an appropriate type (7.9.6.2).

— The result of converting a string to a number by the **atof**, **atoi**, or **atol** function cannot be represented (7.10.1).

— The value of a pointer that refers to space deallocated by a call to the **free** or **realloc** function is referred to (7.10.3).

— The pointer argument to the **free** or **realloc** function does not match a pointer earlier returned by **calloc**, **malloc**, or **realloc**, or the object pointed to has been deallocated by a call to **free** or **realloc** (7.10.3).

— A program executes more than one call to the **exit** function (7.10.4.3).

— The result of an integer arithmetic function (**abs**, **div**, **labs**, or **ldiv**) cannot be represented (7.10.6).

— The shift states for the **mblen**, **mbtowc**, and **wctomb** functions are not explicitly reset to the initial state when the **LC_CTYPE** category of the current locale is changed (7.10.7).

— An array written to by a copying or concatenation function is too small (7.11.2, 7.11.3).

— An invalid conversion specification is found in the format for the **strftime** function (7.12.3.5).

G.3 Implementation-defined behavior

Each implementation shall document its behavior in each of the areas listed in this subclause. The following are implementation-defined:

G.3.1 Translation

— How a diagnostic is identified (5.1.1.3).

G.3.2 Environment

— The semantics of the arguments to **main** (5.1.2.2.1).

— What constitutes an interactive device (5.1.2.3).

G.3.3 Identifiers

— The number of significant initial characters (beyond 31) in an identifier without external linkage (6.1.2).

— The number of significant initial characters (beyond 6) in an identifier with external linkage (6.1.2).

— Whether case distinctions are significant in an identifier with external linkage (6.1.2).

G.3.4 Characters

— The members of the source and execution character sets, except as explicitly specified in this International Standard (5.2.1).

— The shift states used for the encoding of multibyte characters (5.2.1.2).

— The number of bits in a character in the execution character set (5.2.4.2.1).

— The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.1.3.4).

— The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant (6.1.3.4).

— The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (6.1.3.4).

— The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant (6.1.3.4).

— Whether a "plain" **char** has the same range of values as **signed char** or **unsigned char** (6.2.1.1).

G.3.5 Integers

— The representations and sets of values of the various types of integers (6.1.2.5).

— The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (6.2.1.2).

— The results of bitwise operations on signed integers (6.3).

— The sign of the remainder on integer division (6.3.5).

— The result of a right shift of a negative-valued signed integral type (6.3.7).

G.3.6 Floating point

— The representations and sets of values of the various types of floating-point numbers (6.1.2.5).

— The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (6.2.1.3).

— The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number (6.2.1.4).

G.3.7 Arrays and pointers

— The type of integer required to hold the maximum size of an array — that is, the type of the **sizeof** operator, **size_t** (6.3.3.4, 7.1.1).

— The result of casting a pointer to an integer or vice versa (6.3.4).

— The type of integer required to hold the difference between two pointers to elements of the same array, **ptrdiff_t** (6.3.6, 7.1.1).

G.3.8 Registers

— The extent to which objects can actually be placed in registers by use of the **register** storage-class specifier (6.5.1).

G.3.9 Structures, unions, enumerations, and bit-fields

— A member of a union object is accessed using a member of a different type (6.3.2.3).

— The padding and alignment of members of structures (6.5.2.1). This should present no problem unless binary data written by one implementation are read by another.

— Whether a "plain" **int** bit-field is treated as a **signed int** bit-field or as an **unsigned int** bit-field (6.5.2.1).

— The order of allocation of bit-fields within a unit (6.5.2.1).

— Whether a bit-field can straddle a storage-unit boundary (6.5.2.1).

— The integer type chosen to represent the values of an enumeration type (6.5.2.2).

G.3.10 Qualifiers

— What constitutes an access to an object that has volatile-qualified type (6.5.5.3).

G.3.11 Declarators

— The maximum number of declarators that may modify an arithmetic, structure, or union type (6.5.4).

G.3.12 Statements

— The maximum number of **case** values in a **switch** statement (6.6.4.2).

G.3.13 Preprocessing directives

— Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value (6.8.1).

— The method for locating includable source files (6.8.2).

— The support of quoted names for includable source files (6.8.2).

— The mapping of source file character sequences (6.8.2).

— The behavior on each recognized **#pragma** directive (6.8.6).

— The definitions for **__DATE__** and **__TIME__** when respectively, the date and time of translation are not available (6.8.8).

G.3.14 Library functions

— The null pointer constant to which the macro **NULL** expands (7.1.6).

— The diagnostic printed by and the termination behavior of the **assert** function (7.2).

— The sets of characters tested for by the **isalnum**, **isalpha**, **iscntrl**, **islower**, **isprint**, and **isupper** functions (7.3.1).

— The values returned by the mathematics functions on domain errors (7.5.1).

— Whether the mathematics functions set the integer expression **errno** to the value of the macro **ERANGE** on underflow range errors (7.5.1).

— Whether a domain error occurs or zero is returned when the **fmod** function has a second argument of zero (7.5.6.4).

— The set of signals for the **signal** function (7.7.1.1).

— The semantics for each signal recognized by the **signal** function (7.7.1.1).

— The default handling and the handling at program startup for each signal recognized by the **signal** function (7.7.1.1).

— If the equivalent of **signal(sig, SIG_DFL);** is not executed prior to the call of a signal handler, the blocking of the signal that is performed (7.7.1.1).

— Whether the default handling is reset if the **SIGILL** signal is received by a handler specified to the **signal** function (7.7.1.1).

— Whether the last line of a text stream requires a terminating new-line character (7.9.2).

— Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.9.2).

— The number of null characters that may be appended to data written to a binary stream (7.9.2).

— Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file (7.9.3).

— Whether a write on a text stream causes the associated file to be truncated beyond that point (7.9.3).

— The characteristics of file buffering (7.9.3).

— Whether a zero-length file actually exists (7.9.3).

— The rules for composing valid file names (7.9.3).

— Whether the same file can be open multiple times (7.9.3).

— The effect of the **remove** function on an open file (7.9.4.1).

— The effect if a file with the new name exists prior to a call to the **rename** function (7.9.4.2).

— The output for **%p** conversion in the **fprintf** function (7.9.6.1).

— The input for **%p** conversion in the **fscanf** function (7.9.6.2).

— The interpretation of a **-** character that is neither the first nor the last character in the scanlist for **%[** conversion in the **fscanf** function (7.9.6.2).

— The value to which the macro **errno** is set by the **fgetpos** or **ftell** function on failure (7.9.9.1, 7.9.9.4).

— The messages generated by the **perror** function (7.9.10.4).

— The behavior of the **calloc**, **malloc**, or **realloc** function if the size requested is zero (7.10.3).

— The behavior of the **abort** function with regard to open and temporary files (7.10.4.1).

— The status returned by the **exit** function if the value of the argument is other than zero, **EXIT_SUCCESS**, or **EXIT_FAILURE** (7.10.4.3).

— The set of environment names and the method for altering the environment list used by the **getenv** function (7.10.4.4).

— The contents and mode of execution of the string by the **system** function (7.10.4.5).

— The contents of the error message strings returned by the **strerror** function (7.11.6.2).

— The local time zone and Daylight Saving Time (7.12.1).

— The era for the **clock** function (7.12.2.1).

G.4 Locale-specific behavior

The following characteristics of a hosted environment are locale-specific:

— The content of the execution character set, in addition to the required members (5.2.1).

— The direction of printing (5.2.2).

— The decimal-point character (7.1.1).

— The implementation-defined aspects of character testing and case mapping functions (7.3).

— The collation sequence of the execution character set (7.11.4.4).

— The formats for time and date (7.12.3.5).

G.5 Common extensions

The following extensions are widely used in many systems, but are not portable to all implementations. The inclusion of any extension that may cause a strictly conforming program to become invalid renders an implementation nonconforming. Examples of such extensions are new keywords, or library functions declared in standard headers or predefined macros with names that do not begin with an underscore.

G.5.1 Environment arguments

In a hosted environment, the **main** function receives a third argument, **char *envp[]**, that points to a null-terminated array of pointers to **char**, each of which points to a string that provides information about the environment for this execution of the process (5.1.2.2.1).

G.5.2 Specialized identifiers

Characters other than the underscore _, letters, and digits, that are not defined in the required source character set (such as the dollar sign **$**, or characters in national character sets) may appear in an identifier (6.1.2).

G.5.3 Lengths and cases of identifiers

All characters in identifiers (with or without external linkage) are significant and case distinctions are observed (6.1.2).

G.5.4 Scopes of identifiers

A function identifier, or the identifier of an object the declaration of which contains the keyword **extern**, has file scope (6.1.2.1).

G.5.5 Writable string literals

String literals are modifiable. Identical string literals shall be distinct (6.1.4).

G.5.6 Other arithmetic types

Other arithmetic types, such as **long long int**, and their appropriate conversions are defined (6.2.2.1).

G.5.7 Function pointer casts

A pointer to an object or to **void** may be cast to a pointer to a function, allowing data to be invoked as a function (6.3.4). A pointer to a function may be cast to a pointer to an object or to **void**, allowing a function to be inspected or modified (for example, by a debugger) (6.3.4).

G.5.8 Non-**int** bit-field types

Types other than **int**, **unsigned int**, or **signed int** can be declared as bit-fields, with appropriate maximum widths (6.5.2.1).

G.5.9 The **fortran** keyword

The **fortran** declaration specifier may be used in a function declaration to indicate that calls suitable for FORTRAN should be generated, or that different representations for external names are to be generated (6.5.4.3).

G.5.10 The asm keyword

The **asm** keyword may be used to insert assembly language code directly into the translator output. The most common implementation is via a statement of the form

asm (*character-string-literal*) ;

(6.6).

G.5.11 Multiple external definitions

There may be more than one external definition for the identifier of an object, with or without the explicit use of the keyword **extern**, If the definitions disagree, or more than one is initialized, the behavior is undefined (6.7.2).

G.5.12 Empty macro arguments

A macro argument may consist of no preprocessing tokens (6.8.3).

G.5.13 Predefined macro names

Macro names that do not begin with an underscore, describing the translation and execution environments, may be defined by the implementation before translation begins (6.8.8).

G.5.14 Extra arguments for signal handlers

Handlers for specific signals may be called with extra arguments in addition to the signal number (7.7.1.1).

G.5.15 Additional stream types and file-opening modes

Additional mappings from files to streams may be supported (7.9.2), and additional file-opening modes may be specified by characters appended to the **mode** argument of the **fopen** function (7.9.5.3).

G.5.16 Defined file position indicator

The file position indicator is decremented by each successful call to the **ungetc** function for a text stream, except if its value was zero before a call (7.9.7.11).

Anhang B

Benutzerhandbuch

User Manual – Compiler Evaluation Tool

This document shall give an overview of the structure and usage of the compiler evaluation software.

Index

Purpose of the tool: 1
Structure of the programme 1
Usage 3
1. Preparation 3
1.1 C file 3
1.2 Compile Script 3
1.3 Configuration 3
2. Initialization 4
3. Start first test 4
4. Flash Microcontroller and run the 1. test 4
5. Start the second test 4
6. Flash microcontroller and run the 2. test 4
7. Start the third test 4
8. Flash microcontroller and run the 3. test 4
9. Start the fourth test (optional) 5
10. Flash microcontroller and run the 4. test 5
11. Get the final result 5
Folder structure 5
Appendix 7

Purpose of the tool:

This tool analyses C compiler based on the C standard ISO/IEC 9899:1990 (C90, ANSI-C). It evaluates the actual implementation of features which are implementation defined according to the standard. You will find a list of the currently tested features in the appendix.
Some examples are the size of various types, rounding behaviour and the maximum length of identifiers.
The results of these tests will give you a quick overview about the compiler, its settings and the used processor. This information can be used to adjust compiler options, modify already existent code and to configure QA-C without reading the whole handbook.

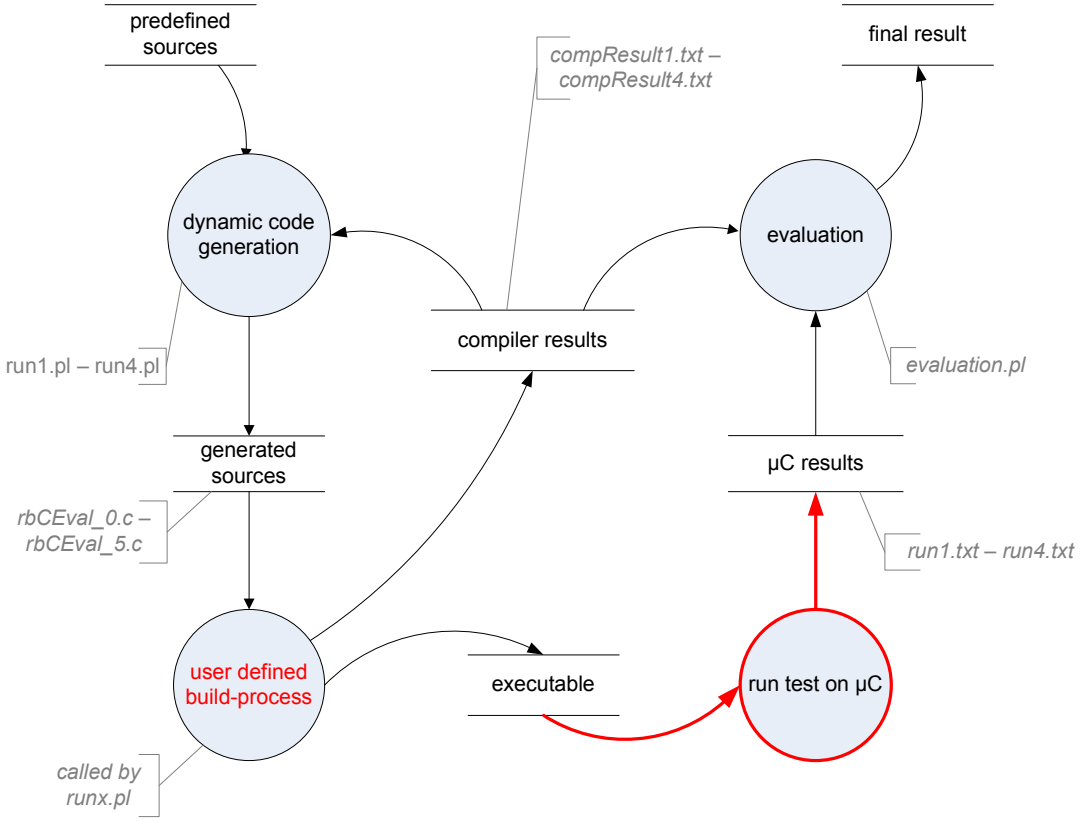
The tool is written in C and Perl. To use it, some basic understanding of both programming languages is recommended. Furthermore you should be familiar with the build process of your project.

Structure of the programme

The Evaluation Tool consists of a bunch of predefined C files and Perl scripts. The Perl scripts control the flow of the programme.
To test the various implementation defined features, it is necessary to have multiple compile processes. For each compile process, the Perl scripts generate the necessary C files into a user defined folder under the names `rbCEval_0.c,... rbCEval_5.c`. It then calls the user defined compile script and waits until it is finished. If the compile and link process was successful and only then, the compile script has to return the success status by returning a text

file. The directories where the files shall be placed can be configured in the file `rbCEval_config.pm`. The Perl script then checks if the text file exists. This is a very general way of checking if the files did compile/link, because the evaluation tool cannot parse the compiler/linker output. The information is read, stored and the text file will be deleted before the next run.
Now, the next set of C files is generated and the process is repeated.
Each script stores the compiler results into a text file `compResult_x.txt` so they won't be lost for further evaluation.
Each of the currently 4 Perl scripts generates an executable at its end which has to be flashed onto the microcontroller for run-time tests. The output of the onboard test has to be stored into `run1.txt` for the first test, `run2.txt` for the second, etc. These text files will be stored in a temporary folder which has to be defined in the file `rbCEval_config.pm`, too.
To get the output from the microcontroller the user has to implement the C-function `rbCEval_outputByte(unsigned char)`. It has to ensure that the passed parameter is stored somehow or transferred to the PC for further usage.
The next script will read the outputs of the previous one(s) and the whole process is repeated.
After all tests are finished, the script `evaluation.pl` is called which will collect all results of the compile, link and onboard tests and generate a readable result.

Dataflow of the programme:



Red: Has to done somehow by the user.

Usage

1. Preparation

Before the Evaluation Tool can be used, you have to provide a **compile script** and at least one additional **C file**.

1.1 C file

Implements a start-up function (eg. `main()`) which calls the function `rbCEval_toolKit()`.
To call this function the header `rbCEval.h` has to be included.
Implement the function `void rbCEval_outputByte(unsigned char)`
this function has to take an unsigned character and store it somehow. Depending on the environment this can be done via the serial port, CAN messages, into an EEPROM or anything else. It doesn't matter how the data are stored as long as they can be read later and then stored into the according text-file.
Each record will be one line of the format:

name=hexvalue

eg: `a1=1C`

Without the content of the macros, the result to be transferred will have a total size of around 1000 Bytes.

1.2 Compile Script

The compile script can be a make file, a Perl script, a batch file or anything else as long as it does the following things and is callable via the command line.

- Generate object files of the following C files:
  - `rbCEval_0.c`
  - `rbCEval_1.c`
  - `rbCEval_2.c`
  - `rbCEval_3.c`
  - `rbCEval_4.c`
  - `rbCEval_5.c`
  - all user defined files which are necessary for an executable program
- After a successful compilation, the object files and the necessary libraries have to be linked to an executable file which can be flashed onto the microcontroller (eg. an `.elf` file). If this executable was generated the script has to **create a text file** in the temporary folder: `rbCEval_compile.txt`

There is an example script for the gcc compiler in the example folder which is written in Perl (`compile.pl`).

1.3 Configuration

The main configuration file for the Perl scripts is the file `rbCEval_config.pm`.
It is possible to place it somewhere else, like a project folder. If a Perl script is called, it has to be done like

`perl -I<path to rbCEval_config.pm> script [options]`

while the path has to be absolute or relative to the main directory `CEval/`.

Here you have to define the following:

`$compileSkript`	Command to start the user defined compile script, batch file, make-file, etc
`$targetPath`	Path where the C source files shall be generated in, similar to where the project specific sources are so that the compile script finds the C source files.
`$includePath`	Path where the include files shall be copied to
`$tmpPath`	Temporary folder for temporary generated files
`$DD`	Directory delimiter ('/' for Unix, Linux; '\\' for Windows)

All paths must be absolute or relative to the main folder of the tool (`CEval/`)

2. Initialization

```
perl init.pl
```

This initialization script will delete all previous results stored in temporary files, check the configured paths and perform some basic tests on the user defined compile script.
If these are successful, the real evaluation can start.
If it fails there will be an error message with more detail. Usually it means that the paths are not correct or the compile script is not working correctly.

3. Start first test

```
perl run1.pl [-f]
```

To avoid any tests about floating-point features, the option `-f` can be used. Once used, it must be used for all test, otherwise there might be inconsistencies.

4. Flash Microcontroller and run the 1. test

The next step is to run the first test on the Microcontroller. The user shall do that manually and must ensure the results of the test are written into the file `run1.txt`

5. Start the second test

```
perl run2.pl [-f]
```

This test contains only one function and is about modulo with zero. This is done with a volatile variable, so it should compile.

6. Flash microcontroller and run the 2. test

Run the third test on the Microcontroller. The results of the test must be written into the file `run2.txt`
Because of a special verification the test can cause a runtime error and the processor might crash. If that happens, the file `run2.txt` still has to be created, but left empty!

7. Start the third test

```
perl > run3.pl [-f]
```

This test contains only one function and is about division with zero. This is done with a volatile variable, so it should compile.

8. Flash microcontroller and run the 3. test

Run the third test on the Microcontroller. The results of the test must be written into the file `run3.txt`

Because of a division with zero the test can cause a runtime error and the processor might crash. If that happens, the file `run3.txt` still has to be created, but left empty!

9. Start the fourth test (optional)

```
Perl -I run4.pl <macros> [<includes>]
```

`macros`	file with macro names to be checked, one macro per line
`includes`	file with include commands (`#include "Header.h"`), if certain headers should be included for the macro test (optional)

This test will give you the content of the macros you pass in a text file to the script (each macro in a separate line). The test is optional.

10. Flash microcontroller and run the 4. test

Run the third test on the Microcontroller. The results of the test must be written into the file `run4.txt`

11. Get the final result

```
perl evaluation.pl [-f]
```

This script will collect all results and display them in a readable format on stdout. For further explanation read the file documentation.

Folder structure

```
CEval
├──admin
├──base
│   └──c_sourcen
├──docs
└──example
    ├──include
    ├──source
    ├──tmp
    ├──obj
    └──out
```

`CEval/`

Main folder of the tool. It contains the Perl scripts, used Perl modules.

1. `init.pl` – Perl script for initialization
2. `run1.pl` – Perl script for the first run
3. `run2.pl` – Perl script for the second run
4. `run3.pl` – Perl script for the third run
5. `run4.pl` – Perl script for the fourth run (optional)
6. `evaluation.pl` – Perl script to produce the final result

7. `rbCEval_config.pm` – main configuration file.
8. `rbCEval_vars.pm` – Perl module where global variables and file names are defined
9. `rbCEval_subroutines` – Perl module used by the start scripts
10. `rbCEval_const.pm` – automatically generated Perl module with the integer constants

`admin/`

- `const.pl` – script to generate `base/c_sourcen/rbCEval_const.h` and `rbCEval_const.pm`
- `constants.csv` – list of integer constants with their appropriate name.
  the Perl script generates a header file and a Perl module from this file, so that the constants for the C programme are the same as used in the final evaluation script.

`base/c_sourcen/`

Contains all predefined C files necessary for the compiler evaluation. Most of the files contain only one function. This helps to check each function separately if it compiles. The appendix contains a list of files and which predefined function(s) it contains.

`docs/`

Folder where this and other documentation about the tool can be found

`example/`

Folder with an example.

- `comile.pl` – a predefined compile script which uses the GNU compiler (gcc). It will generate executables for the current machine it is running on (ie. the PC)

The subfolders in this directory are used by the compile script and for the example evaluation. They are also used in the example configuration file.

Appendix

List of the evaluated features

Length of Byte	in bit
Negative representation	two's complement, one's complement, sign bit
Right shift	logical or arithmetical
Endianess	little endian, big endian
Maximum length of identifiers	up to 1023 digits
Plain `char`	signed or unsigned
Character set of execution environment	ASCII, extended ASCII, no ASCII
Character set during preprocessing	ASCII or not
// comments	accepted or not
Size of data types in bits	(void), char, wchar, shor, int , long, (long long), float, double, long double, sizeof, data pointer, function pointer, pointer difference
Alignment of data types	char, short, int, long, (long long), float, double, long double, data pointer, function pointer
Size and sign of enumerations	various enumerated lists
Padding in structs	fill bytes used or not
Modulo and Division of integer values	• with negative operands • during preprocessing • with zero (constant) • with zero, during run-time (volatile variable)
Casts to smaller signed type	change of bit pattern?
Floating Point	• base • mantissa (not normalized, hence one bit more than internally stored) • epsilon to 1 (smallest difference to 1)
Rounding behaviour of floating point types	• cast from integer to float • if a decimal value is not representable (like 0.1)
Bit-fields	• padding • alignment • bit-order • type of bit fields (char, short, long, long long (int is standard))
Macros (optional)	content of preprocessing macros

Anhang C

Ergebnisse

1. Analyseergebnis
   Mikorprozessor: MC9S12DP512 (HC12, Freescale)
   Compiler : Cosmic HC(S)12 C-Compiler v4.6

2. Vergleich zweier Analyseergebnisse mit dem Programm „Beyond Compare“, einem Vergleichstool, dass die Unterschiede farbig markiert.
   Mikorprozessor: MC9S12DP512 (HC12, Freescale)
   Compiler 1: Cosmic HC(S)12 C-Compiler v4.6
   Compiler 2: Metrowerks CodeWarrior v4.6

```
------------------ Evaluation ---------------------
Date:          2007-09-10
Compilescript: TestEnvBuild.bat
---------------------------------------------------

Identifier:
  The maximum length for an external identifier is 254 characters, (tested up to 1023)
  External identifiers are case sensitive.
  Macro names can have a length of 1023, (tested up to 1023)

Negative Representation is:
  two's complement

Right shift is done:
  arithmetic (dependent on sign)

Endianess:
  big endian, byte order: high byte first (e.g. 4 3 2 1)

Number of Bits in one Byte (char): 8

Plain char is interpreted as:
  unsigned char

The character set of the execution environment is
  ASCII - The complete ASCII set is usable (extended ASCII).

Standard charset for preprocessing is implemented in
  ASCII

C++ style comments (// comment)
  are accepted

The various types have the following size in bits:
    void:           not known
    char:           8
    wchar:          16
    short:          16
    int:            16
    long:           32
    longlong:       not defined
    float:          32
    double:         64
    longdouble:     64
    sizeof:         16
    pointer:        16
    pointerDiff:    16
    functionPointer:16
  There is an integerdatatype with more than 32 bits.

The various types have the following alignment:
    char:           1
    short:          1
    int:            1
```

```
    long:           1
    longlong:       not defined
    float:          1
    double:         1
    longdouble:     1
    pointer:        1
    functionPointer:1

Enumerations, size in bytes and signedness (1 = signed , 0 = unsigned)
  type                              size sign sizeof(a)
  {a,            b,            c}: 1    0     2
  {a=-1,         b,            c}: 1    1     2
  {a=128,        b,            c}: 1    0     2
  {a=-1,         b=128,        c}: 2    1     2
  {a=256,        b,            c}: 2    0     2
  {a=32768,      b,            c}: 2    0     2
  {a=-1,         b= 32768,     c}: 4    1     4
  {a=65536,      b,            c}: 4    0     4
  {a=2147483648, b,            c}: 4    1     4
  {a=-1,         b=2147483648, c}: 4    1     4
  {a=4294967296, b,            c}: not possible!

Padding in Structs with data types of different size:
  fill bytes behind char if char comes before long (char & long): false
  fill bytes behind char if long comes before char (long & char): false

Modulo and Division operations of integer values:
If a/b is representable: (a/b)*b + a%b == a. true
  The result of Integer Division with negative Operands is
    truncated (9/-5=-1, -9/5=-1, -9/-5=1)
  The result of Modulo with negative Operands is negative if
    dividend is negative (9%-5=4, -9%5=-4, -9%-5=-4)

  Integer Division with negative Operands during preprocessing is
    truncated (rounded towards zero)
  The result of Modulo with negative Operands during preproc. is negative if
    dividend is negative (9%-5=4, -9%5=-4, -9%-5=-4)
  Division with zero (constant):
    compiler error

  Modulo with zero (constant):
    compiler error
  Division with zero during run-time(volatile var).
    result:   zero (3/0 = 0)

  Modulo with zero during run-time(volatile var)
    result:   dividend (3%0=3)

Casting to a smaller signed type:
  unsigned char -> signed char:  no bits changed
  signed long   -> signed char:  no bits changed
  unsigned long -> signed char:  no bits changed

Floating Point:
```

```
    Base = 2
    precision (mantissa) of float:       23 bits
    precision (mantissa) of double:      52 bits
    precision (mantissa) of long double: 52 bits
  exponent = size in Bits - precision - signbit
    exponent of float:                   8 bits
    exponent of double:                  11 bits
    exponent of long double:             11 bits
  representable minimal difference (epsilon) to 1:
  (it might be smaller)
    epsilon for float:                   ca. 10^-6
    epsilon for double:                  ca. 10^-15
    epsilon for long double:             ca. 10^-15
  Rounding behaviour if the value is not representable
    Fractional part (eg. 0.1) is rounded:  to nearest
    Cast Integer->Float is rounded:        to zero

Bit-fields:
  Padding:
    no padding bits used (e.g. int 16 Bit: 10+16+6 = 32 Bit => 2 int bit-fields)
  Order of bits in a complete int bit-field: LSB first
  WARNING: Size of bit-field differs according to used bits!!
    WARNING: First bit in bit-fields of various size is not at the same position!
  Signedness of int bit-fields:
    each bit-field is unsigned, regardless of its definition (eg.: signed int bf:4 has a range of [0,15])
  WARNING, implementation doesn't match C90!
  Allowed types of bit-fields (appart from int)
    char, short, long,

Content of predefined macros:
  __STDC__=1
  NULL=(void *)0
  RBCEVAL_H=
  __FILE__="c:\\projects\\CEval\\example\\source\\rbCEval_4.c"
  __LINE__=31
----------------------- END -----------------------
```

Modus: Alle Zeilen

Linke Datei: C:\projects\CEval\result\hc12_cosmic_070910.txt
Rechte Datei: C:\projects\CEval\result\hc12_metro_070910.txt

------------------ Evaluation ----------------------	=	------------------ Evaluation ----------------------
Date: 2007-09-10		Date: 2007-09-10
Compilescript: TestEnvBuild.bat	<>	Compilescript: freescale.bat
--	=	--
Identifier:		Identifier:
The maximum length for an external identifier is 254 characters, (test » ed up to 1023)	<>	The maximum length for an external identifier is 1023 characters, (tes » ted up to 1023)
External identifiers are case sensitive.	=	External identifiers are case sensitive.
Macro names can have a length of 1023, (tested up to 1023)		Macro names can have a length of 1023, (tested up to 1023)
Negative Representation is:		Negative Representation is:
two's complement		two's complement
Right shift is done:		Right shift is done:
arithmetic (dependent on sign)		arithmetic (dependent on sign)
Endianess:		Endianess:
big endian, byte order: high byte first (e.g. 4 3 2 1)		big endian, byte order: high byte first (e.g. 4 3 2 1)
Number of Bits in one Byte (char): 8		Number of Bits in one Byte (char): 8
Plain char is interpreted as:		Plain char is interpreted as:
unsigned char	<>	signed char
	=	
The character set of the execution environment is		The character set of the execution environment is
ASCII - The complete ASCII set is usable (extended ASCII).		ASCII - The complete ASCII set is usable (extended ASCII).
Standard charset for preprocessing is implemented in		Standard charset for preprocessing is implemented in
ASCII		ASCII
C++ style comments (// comment)		C++ style comments (// comment)
are accepted		are accepted
The various types have the following size in bits:		The various types have the following size in bits:
void: not known		void: not known
char: 8		char: 8

Linke Datei: C:\projects\CEval\result\hc12_cosmic_070910.txt
Rechte Datei: C:\projects\CEval\result\hc12_metro_070910.txt
(Fortsetzung)

wchar: 16	<>	wchar: 8
short: 16 int: 16 long: 32	=	short: 16 int: 16 long: 32
longlong: not defined	<>	longlong: 32
float: 32	=	float: 32
double: 64 longdouble: 64	<>	double: 32 longdouble: 32
sizeof: 16 pointer: 16 pointerDiff: 16 functionPointer:16 There is an integerdatatype with more than 32 bits. The various types have the following alignment: char: 1 short: 1 int: 1 long: 1	=	sizeof: 16 pointer: 16 pointerDiff: 16 functionPointer:16 There is an integerdatatype with more than 32 bits. The various types have the following alignment: char: 1 short: 1 int: 1 long: 1
longlong: not defined	<>	longlong: 4
float: 1 double: 1 longdouble: 1 pointer: 1 functionPointer:1 Enumerations, size in bytes and signedness (1 = signed , 0 = unsigned) type size sign sizeof(a)	=	float: 1 double: 1 longdouble: 1 pointer: 1 functionPointer:1 Enumerations, size in bytes and signedness (1 = signed , 0 = unsigned) type size sign sizeof(a)
{a, b, c}: 1 0 2	<>	{a, b, c}: 2 1 2
{a=-1, b, c}: 1 1 2		{a=-1, b, c}: 2 1 2
{a=128, b, c}: 1 0 2		{a=128, b, c}: 2 1 2
{a=-1, b=128, c}: 2 1 2	=	{a=-1, b=128, c}: 2 1 2
{a=256, b, c}: 2 0 2	<>	{a=256, b, c}: 2 1 2
{a=32768, b, c}: 2 0 2		{a=32768, b, c}: not possible!
{a=-1, b= 32768, c}: 4 1 4		{a=-1, b= 32768, c}: not possible!
{a=65536, b, c}: 4 0 4		{a=65536, b, c}: not possible!
{a=2147483648, b, c}: 4 1 4		{a=2147483648, b, c}: not possible!
{a=-1, b=2147483648, c}: 4 1 4		{a=-1, b=2147483648, c}: not possible!

Linke Datei: C:\projects\CEval\result\hc12_cosmic_070910.txt
Rechte Datei: C:\projects\CEval\result\hc12_metro_070910.txt
(Fortsetzung)

{a=4294967296, b, c}: not possible!	=	{a=4294967296, b, c}: not possible!
Padding in Structs with data types of different size:		Padding in Structs with data types of different size:
fill bytes behind char if char comes before long (char & long): false		fill bytes behind char if char comes before long (char & long): false
fill bytes behind char if long comes before char (long & char): false		fill bytes behind char if long comes before char (long & char): false
Modulo and Division operations of integer values:		Modulo and Division operations of integer values:
If a/b is representable: (a/b)*b + a%b == a. true		If a/b is representable: (a/b)*b + a%b == a. true
The result of Integer Division with negative Operands is		The result of Integer Division with negative Operands is
truncated (9/-5=-1, -9/5=-1, -9/-5=1)		truncated (9/-5=-1, -9/5=-1, -9/-5=1)
The result of Modulo with negative Operands is negative if		The result of Modulo with negative Operands is negative if
dividend is negative (9%-5=4, -9%5=-4, -9%-5=-4)		dividend is negative (9%-5=4, -9%5=-4, -9%-5=-4)
Integer Division with negative Operands during preprocessing is		Integer Division with negative Operands during preprocessing is
truncated (rounded towards zero)		truncated (rounded towards zero)
The result of Modulo with negative Operands during preproc. is negativ		The result of Modulo with negative Operands during preproc. is negativ
» e if		» e if
dividend is negative (9%-5=4, -9%5=-4, -9%-5=-4)		dividend is negative (9%-5=4, -9%5=-4, -9%-5=-4)
Division with zero (constant):		Division with zero (constant):
compiler error		compiler error
Modulo with zero (constant):		Modulo with zero (constant):
compiler error		compiler error
Division with zero during run-time(volatile var).		Division with zero during run-time(volatile var).
result: zero (3/0 = 0)		result: zero (3/0 = 0)
Modulo with zero during run-time(volatile var)		Modulo with zero during run-time(volatile var)
result: dividend (3%0=3)		result: dividend (3%0=3)
Casting to a smaller signed type:		Casting to a smaller signed type:
unsigned char -> signed char: no bits changed		unsigned char -> signed char: no bits changed
signed long -> signed char: no bits changed		signed long -> signed char: no bits changed
unsigned long -> signed char: no bits changed		unsigned long -> signed char: no bits changed
Floating Point:		Floating Point:
Base = 2		Base = 2
precision (mantissa) of float: 23 bits	<>	precision (mantissa) of float: 22 bits
precision (mantissa) of double: 52 bits		precision (mantissa) of double: 22 bits

Linke Datei: C:\projects\CEval\result\hc12_cosmic_070910.txt
Rechte Datei: C:\projects\CEval\result\hc12_metro_070910.txt
(Fortsetzung)

precision (mantissa) of long double: 52 bits		precision (mantissa) of long double: 22 bits
exponent = size in Bits - precision - signbit	=	exponent = size in Bits - precision - signbit
exponent of float: 8 bits	<>	exponent of float: 9 bits
exponent of double: 11 bits		exponent of double: 9 bits
exponent of long double: 11 bits		exponent of long double: 9 bits
representable minimal difference (epsilon) to 1:	=	representable minimal difference (epsilon) to 1:
(it might be smaller)		(it might be smaller)
epsilon for float: ca. 10^-6		epsilon for float: ca. 10^-6
epsilon for double: ca. 10^-15	<>	epsilon for double: ca. 10^-6
epsilon for long double: ca. 10^-15		epsilon for long double: ca. 10^-6
Rounding behaviour if the value is not representable	=	Rounding behaviour if the value is not representable
Fractional part (eg. 0.1) is rounded: to nearest		Fractional part (eg. 0.1) is rounded: to nearest
Cast Integer->Float is rounded: to zero		Cast Integer->Float is rounded: to zero
Bit-fields:		Bit-fields:
Padding:		Padding:
no padding bits used (e.g. int 16 Bit: 10+16+6 = 32 Bit => 2 int bit » -fields)		no padding bits used (e.g. int 16 Bit: 10+16+6 = 32 Bit => 2 int bit » -fields)
Order of bits in a complete int bit-field: LSB first		Order of bits in a complete int bit-field: LSB first
WARNING: Size of bit-field differs according to used bits!!		WARNING: Size of bit-field differs according to used bits!!
WARNING: First bit in bit-fields of various size is not at the same » position!		WARNING: First bit in bit-fields of various size is not at the same » position!
Signedness of int bit-fields:		Signedness of int bit-fields:
each bit-field is unsigned, regardless of its definition (eg.: signe » d int bf:4 has a range of [0,15])	<>	plain int bit-field is signed, the others as defined
WARNING, implementation doesn't match C90!		
Allowed types of bit-fields (appart from int)	=	Allowed types of bit-fields (appart from int)
char, short, long,		char, short, long,
Content of predefined macros:		Content of predefined macros:
__STDC__=1	<>	__STDC__=0
NULL=(void *)0		NULL=((void *) 0)
RBCEVAL_H=	=	RBCEVAL_H=
__FILE__="c:\\projects\\CEval\\example\\source\\rbCEval_4.c"	<>	__FILE__="c:\projects\CEval\example\source\rbCEval_4.c"
__LINE__=31	=	__LINE__=31
---------- END ----------		---------- END ----------